

Поиск на основе искусственного интеллекта

Трей Грейнджер
Дуг Тернбулл
Макс Ирвин



Треј Грейнджер, Дуг Тернбулл, Макс Ирвин
Предисловие Гранта Ингерсолла

*Поиск на основе
искусственного интеллекта*

AI-Powered Search

TREY GRAINGER
DOUG TURNBULL
MAX IRWIN

Foreword by GRANT INGERSOLL



MANNING
Shelter Island

Поиск на основе искусственного интеллекта

ТРЕЙ ГРЕЙНДЖЕР
ДУГ ТЕРНБУЛЛ
МАКС ИРВИН
Предисловие ГРАНТА ИНГЕРСОЛЛА



Москва, 2025

УДК 004.738.52:004.8

ББК 16.263

Г79

Трей Грейнджер, Дуг Тернбулл, Макс Ирвин

Г79 Поиск на основе искусственного интеллекта / пер. с англ. И. Л. Люско. – М.: ДМК Пресс, 2025. – 586 с.: ил.

ISBN 978-5-93700-180-1

Современные поисковые системы выходят далеко за рамки простого сопоставления поисковых запросов с базой данных. Прочитав эту книгу, вы получите знания и навыки, необходимые для разработки продвинутых поисковых приложений на основе ИИ, способных автоматически обучаться на основе каждого обновления контента и взаимодействия с пользователем. Ключевые понятия и методы проиллюстрированы доступными для понимания примерами.

Издание будет полезно специалистам, занятым разработкой и внедрением поисковых систем различного масштаба – от корпоративного до глобального, а также всем желающим повысить профессиональную эрудицию.

УДК 004.738.52:004.8

ББК 16.263

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

©DMKPress 2025. Authorized translation of the English edition. © 2025 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN (англ.) 978-1617296970
ISBN (рус.) 978-5-93700-180-1

©2025 by Trey Grainger. All rights reserved
© Оформление, издание, перевод, ДМК
Пресс, 2025

Краткое оглавление

ЧАСТЬ 1. Современная релевантность поиска	29
1 ■ <i>Знакомство с поиском на основе ИИ</i>	<i>31</i>
2 ■ <i>Работа с естественным языком</i>	<i>58</i>
3 ■ <i>Ранжирование и релевантность на основе контента</i>	<i>87</i>
4 ■ <i>Краудсорсинговая релевантность</i>	<i>120</i>
ЧАСТЬ 2. Изучение домен-специфичного намерения	147
5 ■ <i>Что такое графы знаний</i>	<i>149</i>
6 ■ <i>Использование контекста для изучения домен-специфичного языка</i>	<i>214</i>
7 ■ <i>Интерпретация намерения запроса через семантический поиск</i>	<i>215</i>
ЧАСТЬ 3. Отраженный интеллект	249
8 ■ <i>Модели бустинга сигналов</i>	<i>251</i>
9 ■ <i>Персонализированный поиск</i>	<i>278</i>
10 ■ <i>Обучение ранжированию для обобщаемой релевантности поиска</i>	<i>356</i>
11 ■ <i>Автоматизация обучения ранжированию с помощью моделей кликов</i>	<i>357</i>
12 ■ <i>Преодоление предвзятости ранжирования с помощью активного обучения.....</i>	<i>390</i>
ЧАСТЬ 4. Передний край поиска.....	421
13 ■ <i>Семантический поиск с плотными векторами</i>	<i>423</i>
14 ■ <i>Ответы на вопросы с помощью тонко настроенной большой языковой модели.....</i>	<i>486</i>
15 ■ <i>Базовые модели и новые парадигмы поиска</i>	<i>520</i>

Оглавление

https://t.me/it_books/2

Предисловие от издательства	13
Введение	14
Предисловие	16
Благодарности	17
Об этой книге	19
О авторах	26
Об иллюстрации на обложке	27
ЧАСТЬ 1. Современная релевантность поиска	29
1 Знакомство с поиском на основе ИИ	31
1.1. Что такое поиск на основе ИИ?	34
1.2. Что такое намерения пользователя	38
1.2.1. Что такое поисковая система?	39
1.2.2. Что предлагают рекомендательные системы	40
1.2.3. Спектр персонализации между поиском и рекомендациями	41
1.2.4. Семантический поиск и графы знаний	43
1.2.5. Что такое измерения намерения пользователя	44
1.3. Как работает поиск на основе ИИ?	45
1.3.1. Основа поиска	47
1.3.2. Отраженный интеллект в петлях обратной связи	47
1.3.3. Бустинг сигналов, совместная фильтрация и обучение ранжированию	48
1.3.4. Интеллект контента и предметной области	50
1.3.5. Генеративный ИИ и RAG	52
1.3.6. Контролируемый искусственный интеллект в сравнении с искусственным интеллектом «черного ящика»	54
1.3.7. Архитектура поисковой системы на основе ИИ	54
Резюме	56
2 Работа с естественным языком	58
2.1. Миф о неструктурированных данных	59
2.1.1. Типы неструктурированных данных	60
2.1.2. Типы данных в традиционных структурированных базах данных	61
2.1.3. Объединения, нечеткие объединения и разрешение сущностей в неструктурированных данных	63
2.2. Структура естественного языка	68
2.3. Распределительная семантика и эмбединги	70
2.4. Моделирование домен-специфичных знаний	76
2.5. Проблемы понимания естественного языка для поиска	79
2.5.1. Проблема неоднозначности (полисемия)	80
2.5.2. Проблема понимания контекста	80
2.5.3. Проблема персонализации	81
2.5.4. Проблемы интерпретации запросов по сравнению с интерпретацией документов	82
2.5.5. Проблемы интерпретации намерения запроса	82
2.6. Контент + сигналы: топливо, питающее поиск на основе ИИ	84
Резюме	85

3	<i>Ранжирование и релевантность на основе контента</i>	<i>87</i>
3.1.	Оценка векторов запросов и документов с помощью косинусного сходства	88
3.1.1.	Преобразование текста в векторы	89
3.1.2.	Вычисление сходства между плотными векторными представлениями.....	90
3.1.3.	Расчет сходства между разреженными векторными представлениями.....	91
3.1.4.	Частота термина: измерение того, насколько хорошо документы соответствуют термину.....	94
3.1.5.	Обратная частота документа: измерение важности термина в запросе	99
3.1.6.	TF-IDF: сбалансированная метрика взвешивания для текстовой релевантности	101
3.2.	Управление расчетом релевантности.....	102
3.2.1.	BM25: стандартный алгоритм сходства текста по умолчанию	102
3.2.2.	Функции, функции, везде!	107
3.2.3.	Выбор мультипликативного или аддитивного бустинга для функций релевантности.....	111
3.2.4.	Различение сопоставления (фильтрации) и ранжирования (оценки) документов.....	112
3.2.5.	Логическое соответствие: взвешивание взаимосвязей между терминами в запросе	114
3.2.6.	Разделение задач: фильтрация и оценка	116
3.3.	Реализация пользовательского и домен-специфичного ранжирования релевантности.....	118
	Резюме	119
4	<i>Краудсорсинговая релевантность</i>	<i>120</i>
4.1.	Работа с пользовательскими сигналами	121
4.1.1.	Контент, сигналы и модели	121
4.1.2.	Настройка наших наборов данных о продуктах и сигналах (RetroTech)	123
4.1.3.	Изучение данных сигналов.....	127
4.1.4.	Моделирование пользователей, сеансов и запросов	129
4.2.	Знакомство с отраженным интеллектом.....	130
4.2.1.	Что такое отраженный интеллект?.....	131
4.2.2.	Популяризированная релевантность посредством бустинга сигналов	132
4.2.3.	Персонализированная релевантность через совместную фильтрацию	138
4.2.4.	Обобщенная релевантность через обучение ранжированию ..	140
4.2.5.	Другие модели отраженного интеллекта	142
4.2.6.	Краудсорсинг из контента	143
	Резюме	145
	ЧАСТЬ 2. Изучение домен-специфичного намерения.....	147
5	<i>Что такое графы знаний</i>	<i>149</i>
5.1.	Работа с графами знаний	150
5.2.	Использование нашей поисковой системы в качестве графа знаний....	152

5.3. Автоматическое извлечение графов знаний из контента.....	152
5.3.1. Извлечение произвольных отношений из текста	153
5.3.2. Извлечение гипонимов и гиперонимов из текста.....	156
5.4. Изучение намерений путем обхода семантических графов знаний	159
5.4.1. Что такое семантический граф знаний?	159
5.4.2. Индексирование наборов данных.....	161
5.4.3. Структура SKG	161
5.4.4. Расчет весов ребер для измерения связанности узлов	164
5.4.5. Использование SKG для расширения запроса	168
5.4.6. Использование SKG для рекомендаций на основе контента.....	173
5.4.7. Использование SKG для моделирования произвольных отношений.....	176
5.5. Использование графов знаний для семантического поиска	179
Резюме	179

6 Использование контекста для изучения домен-специфичного языка181

6.1. Классификация намерения запроса	182
6.2. Устранение неоднозначности смысла запроса	185
6.3. Изучение связанных фраз из сигналов запроса.....	191
6.3.1. Просмотр журналов запросов для поиска связанных запросов	192
6.3.2. Поиск связанных запросов через взаимодействие продуктов	198
6.4. Обнаружение фраз из пользовательских сигналов	203
6.4.1. Обработка запросов как сущностей.....	204
6.4.2. Извлечение сущностей из более сложных запросов.....	205
6.5. Ошибки и альтернативные представления	205
6.5.1. Изучение исправлений орфографии из документов	206
6.5.2. Изучение исправлений орфографии по сигналам пользователя	207
6.6. Собираем все вместе.....	214
Резюме	214

7 Интерпретация намерения запроса через семантический поиск ... 215

7.1. Механика интерпретации запроса.....	216
7.2. Индексирование и поиск в наборе данных локальных отзывов.....	219
7.3. Пример сквозного семантического поиска	222
7.4. Конвейеры интерпретации запроса.....	224
7.4.1. Парсинг запроса для семантического поиска	224
7.4.2. Обогащение запроса для семантического поиска	234
7.4.3. Разреженные лексические модели и модели расширения.....	240
7.4.4. Преобразование запроса для семантического поиска.....	243
7.4.5. Поиск с помощью семантически улучшенного запроса	245
Резюме	246

ЧАСТЬ 3. Отраженный интеллект249

8 Модели бустинга сигналов251

8.1. Базовый бустинг сигналов.....	252
8.2. Нормирование сигналов.....	253
8.3. Борьба со спамом сигналов	256
8.3.1. Использование спама сигналов для манипулирования результатами поиска.....	256
8.3.2. Борьба со спамом сигналов с помощью фильтрации на основе пользователей.....	259

8.4. Объединение нескольких типов сигналов.....	261
8.5. Временные спады и короткоживущие сигналы	264
8.5.1. Обработка нечувствительных ко времени сигналов.....	265
8.5.2. Обработка сигналов, чувствительных ко времени.....	266
8.6. Бустинг во время индексирования или во время запроса: балансировка масштаба и гибкости	269
8.6.1. Компромиссы при использовании бустинга во время запроса	269
8.6.2. Реализация бустинга сигналов во время индексирования.....	271
8.6.3. Компромиссы при реализации бустинга во время индексирования	274
Резюме	277

9 *Персонализированный поиск*278

9.1. Персонализированный поиск или рекомендации	279
9.1.1. Персонализированные запросы.....	281
9.1.2. Рекомендации, управляемые пользователем	282
9.2. Приближения алгоритмов рекомендаций.....	283
9.2.1. Рекомендации на основе контента.....	283
9.2.2. Рекомендации на основе поведения	284
9.2.3. Мультимодальные рекомендательные системы.....	286
9.3. Реализация совместной фильтрации.....	287
9.3.1. Изучение скрытых признаков пользователя и предмета с помощью матричной факторизации.....	287
9.3.2. Реализация совместной фильтрации с помощью метода чередующихся наименьших квадратов	292
9.3.3. Персонализация результатов поиска с бустингом рекомендаций....	299
9.4. Персонализация поиска с использованием эмбедингов на основе контента	303
9.4.1. Генерация латентных признаков на основе контента	304
9.4.2. Реализация категориальных ограничений для персонализации.....	307
9.4.3. Интеграция персонализации на основе эмбедингов в результаты поиска	313
9.5. Проблемы с персонализацией результатов поиска.....	319
Резюме	321

10 *Обучение ранжированию для обобщаемой релевантности поиска*322

10.1. Что такое LTR?	323
10.1.1. Выход за рамки ручной настройки релевантности.....	323
10.1.2. Реализация LTR в реальном мире	324
10.2. Шаг 1: список суждений, начиная с обучающих данных	327
10.3. Шаг 2: логирование признаков и инжиниринг.....	329
10.3.1. Хранение признаков в современном поисковом движке	330
10.3.2. Логирование признаков из корпуса нашего поискового движка	331
10.4. Шаг 3: преобразование LTR в традиционную задачу машинного обучения	333
10.4.1. SVMrank: преобразование ранжирования в бинарную классификацию	335
10.4.2. Преобразование нашей задачи обучения LTR в бинарную классификацию	337

10.5. Шаг 4: обучение (и тестирование!) модели.....	345
10.5.1. Превращение вектора разделяющей гиперплоскости в оценочную функцию.....	346
10.5.2. Тест-драйв модели	347
10.5.3. Валидация модели.....	348
10.6. Шаги 5 и 6: загрузка модели и поиск.....	350
10.6.1. Запуск и использование модели LTR	350
10.6.2. Примечание о производительности LTR.....	353
10.7. Почистить и повторить	355
Резюме	356

11 Автоматизация обучения ранжированию с помощью моделей кликов.....357

11.1. (Повторное) создание списков суждений из сигналов.....	359
11.1.1. Генерация неявных вероятностных суждений из сигналов.....	360
11.1.2. Обучение модели LTR с использованием вероятностных суждений.....	362
11.1.3. Показатель кликабельности: ваша первая модель кликов	363
11.1.4. Распространенные предвзятости в суждениях	367
11.2. Преодоление предвзятости позиции	368
11.2.1. Определение предвзятости позиции.....	369
11.2.2. Предвзятость позиции в данных RetroTech	369
11.2.3. Упрощенная динамическая байесовская сеть: модель кликов, которая преодолевает предвзятость позиции.....	371
11.3. Управление предвзятостью уверенности: не пересматривать свою модель из-за нескольких удачных кликов	377
11.3.1. Проблема низкой достоверности в данных о кликах	377
11.3.2. Использование бета-приорного распределения для моделирования достоверности вероятностным образом.....	380
11.4. Изучение ваших обучающих данных в системе LTR.....	387
Оценки SDBN с использованием бета-распределения	388
Резюме	389

12 Преодоление предвзятости ранжирования с помощью активного обучения.....390

12.1. Наш автоматизированный движок LTR в нескольких строках кода	392
12.1.1. Превращение кликов в обучающие данные (глава 11 в одной строке кода)	393
12.1.2. Обучение и оценка модели в нескольких вызовах функций.....	395
12.2. А/В-тестирование новой модели.....	397
12.2.1. Выбираем лучшую модель для тестирования	397
12.2.2. Определение А/В-теста в контексте автоматизированного LTR	398
12.2.3. Перевод лучшей модели в А/В-тест.....	399
12.2.4. Когда «хорошие» модели работают плохо: чему мы можем научиться из неудачного А/В-теста.....	401
12.3. Преодоление предвзятости представления: знание того, когда исследовать, а когда эксплуатировать	403
12.3.1. Предвзятость представления в обучающих данных RetroTech	405
12.3.2. За пределами ad hoc: вдумчивое исследование с помощью гауссова процесса	407
12.3.3. Изучение результата наших исследований.....	414

12.4. Разработка, исследование, сбор, сортировка, повторение: надежный автоматизированный цикл LTR	417
Резюме	419
ЧАСТЬ 4. Передний край поиска.....	421
13 Семантический поиск с плотными векторами	423
13.1. Представление смысла посредством эмбедингов.....	424
13.2. Поиск с использованием плотных векторов	426
13.2.1. Краткая информация о разреженных векторах	426
13.2.2. Система поиска по плотным векторам.....	427
13.3. Получение текстовых эмбедингов с помощью трансформер-кодировщика.....	432
13.3.1. Что такое трансформер?.....	432
13.3.2. Открытые предобученные модели трансформеров	434
13.4. Применение трансформеров для поиска.....	435
13.4.1. Использование набора данных Stack Exchange outdoors	436
13.4.2. Тонкая настройка и семантический анализ сходства текста....	439
13.4.3. Знакомство с библиотекой трансформеров SBERT	440
13.5. Автозаполнение естественного языка.....	443
13.5.1. Получение фраз существительных и глаголов для нашего словаря ближайшего соседа	444
13.5.2. Получение эмбедингов.....	446
13.5.3. Приближенный поиск ближайших соседей	451
13.5.4. Реализация индекса ANN.....	453
13.6. Семантический поиск с эмбедингами LLM	456
13.6.1. Получение заголовков и их эмбедингов.....	457
13.6.2. Создание и поиск индекса ближайшего соседа.....	458
13.7. Квантизация и обучение представлениям для более эффективного векторного поиска	461
13.7.1. Скалярная квантизация	463
13.7.2. Бинарная квантизация	470
13.7.3. Продуктовая квантизация	472
13.7.4. Репрезентативное обучение Matryoshka	475
13.7.5. Объединение нескольких методов оптимизации векторного поиска	478
13.8. Кросс-кодировщики и би-кодировщики – сравнение	481
Резюме	485
14 Ответы на вопросы с помощью тонко настроенной большой языковой модели	486
14.1. Обзор модели вопрос–ответ	487
14.1.1. Как работает модель вопрос–ответ	487
14.1.2. Шаблон ретривер–ридер	493
14.2. Создание обучающего набора данных для модели вопрос–ответ	496
14.2.1. Сбор и очистка набора данных вопрос–ответ.....	497
14.2.2. Создание набора silver: автоматическая маркировка данных из предварительно обученной модели	499
14.2.3. Обучение с участием человека: ручная коррекция набора silver для получения набора golden	502
14.2.4. Форматирование набора golden для обучения, тестирования и проверки	503
14.3. Тонкая настройка модели вопрос–ответ	505

14.3.1. Токенизация и формирование наших маркированных данных	507
14.3.2. Настройка модели-тренера	510
14.3.3. Делаем обучение и оцениваем потери	511
14.3.4. Валидация и подтверждение с удерживанием.....	512
14.4. Создание ридера с новой тонко настроенной моделью	513
14.5. Инкорпорация ретривера: использование модели вопрос–ответ с поисковым движком	515
14.5.1. Шаг 1: запрос ретривера.....	515
14.5.2. Шаг 2: вывод ответов из модели ридера	516
14.5.3. Шаг 3: реранкинг ответов.....	517
14.5.4. Шаг 4: возврат результатов путем объединения ретривера, ридера и реранкера.....	518
Резюме	519
15 Базовые модели и новые парадигмы поиска	520
15.1. Что такое базовые модели.....	521
15.1.1. Что можно считать базовой моделью?	522
15.1.2. Обучение, тонкая настройка в сравнении с подсказкой	523
15.2. Генеративный поиск	526
15.2.1. Генерация с дополненной выборкой	529
15.2.2. Суммаризация результатов с использованием базовых моделей...	532
15.2.3. Генерация данных с использованием базовых моделей	535
15.2.4. Оценка генеративного вывода.....	539
15.2.5. Построение вашей собственной метрики	541
15.2.6. Оптимизация алгоритмических подсказок	543
15.3. Мультимодальный поиск	545
15.3.1. Общие режимы мультимодального поиска.....	545
15.3.2. Реализация мультимодального поиска	548
15.4. Другие появляющиеся парадигмы поиска на основе ИИ.....	554
15.4.1. Разговорный и контекстный поиск	554
15.4.2. Поиск на основе агентов.....	556
15.5. Гибридный поиск	557
15.5.1. Алгоритм RRF	557
15.5.2. Другие алгоритмы гибридного поиска.....	564
15.6. Конвергенция контекстных технологий	566
15.7. Все вышеперечисленное, пожалуйста!.....	568
Резюме	568
Приложение А. Запуск примеров кода	570
A.1. Общая структура примеров кода.....	570
A.2. Извлечение исходного кода.....	571
A.3. Сборка и запуск кода	571
A.4. Работа с Jupyter	573
A.5. Работа с Docker	575
Приложение В. Поддерживаемые поисковые системы и векторные базы данных	576
B.1. Поддерживаемые движки	576
B.2. Динамическая замена движка.....	577
B.3. Абстракции движка и коллекции.....	578
B.4. Добавление поддержки для дополнительных движков	579
Предметный указатель	581

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Введение

В течение последних двух десятилетий информационный поиск был в центре почти каждого аспекта нашего технического существования как людей. Нужно найти факт? Выполните поиск. Хотите посетить новый ресторан? Выполните поиск. Нужно найти начало тропы в горах для вашего похода на выходных? Выполните поиск. Однако для многих инженеров основы того, как работает поиск или как он выходит за рамки простого сопоставления ключевых слов, чтобы действительно раскрыть то, что нужно пользователям от информационной системы, являются загадкой, о которой не рассказывают почти ни на каких курсах и учебных сборах по информатике. Учитывая этот относительный недостаток инструкций в новый золотой век ИИ, сейчас самое время дать *поиску на основе ИИ* возможность оставить свой след в мире, научив читателей основным принципам, необходимым для понимания работы ИИ в любом компьютерном приложении.

В основе всех поисковых систем лежит именно это: раскрытие информации, которое помогает пользователям принимать более обоснованные решения, необходимые для ориентирования в своем мире. Это раскрытие происходит в основном четырьмя способами.

1. Перебор данных, поиск соответствующих фрагментов информации, ранжирование и возврат наиболее важных фрагментов для их синтеза пользователем.

2. Обобщение данных в более мелкие, более удобоваримые формы для совместного использования и совместной работы с помощью визуализаций и других абстракций.

3. Соотнесение данных с другими, в идеале знакомыми фрагментами информации и концептами.

4. Подача любого из этих трех способов вместе с другим контекстом от пользователя в большой языковой модели (LLM) для дальнейшего синтеза, суммаризации и освоения при постоянном обновлении на основе отзывов пользователей во взаимодействии с ними.

За эти же два десятилетия, когда поиск в нашей жизни на уровне потребителя стал повсеместным, поисковые системы, управляющие этим миром, такие как Google, Elasticsearch, Apache Solr и др., эволюционировали, чтобы делать не только поиск и ранжирование методами, о которых сказано выше, но и решать другие задачи, и не только для текстовых данных, но и для всех других видов данных.

Поисковые системы продвинулись вперед в решении этих задач путем глубокого внедрения статистического анализа, машинного обучения, больших языковых моделей и обработки естественного языка; другими словами, интегрируя методы искусственного интеллекта

в каждый элемент своего ядра. И все же, несмотря на глубину и охват возможностей поисковых систем, эти методы обычно упускают из виду, как то, что, собственно, и производит «поиск по ключевым словам».

В книге «Поиск на основе искусственного интеллекта» Трей, Дуг и Макс изложили насыщенное и подробное руководство, предназначенное для того, чтобы провести инженеров через все детали создания интеллектуальных информационных систем, используя все доступные средства: LLM, домен-специфичные знания, базы знаний и графы и, наконец, пользовательские и краудсорсинговые сигналы. Примеры в книге иллюстрируют важнейшие понятия доступными и простыми для понимания способами.

Как человек, потративший большую часть своей карьеры на создание, обучение и продвижение поиска как средства, помогающего решать некоторые из самых важных проблем нашего времени, я был свидетелем моментов, которые дают старт и направляют инженеров (после того, как они прорываются через неопределенность, присущую работе с запутанными мультимодальными данными) на построение своей дальнейшей карьеры в одной из самых сложных и интересных областей нашего времени. Я надеюсь, что, прочитав эту книгу, вы тоже найдете бесконечное очарование в мире поиска.

Счастливого поиска!

– Грант Ингерсолл, генеральный директор
и основатель Develomentor LLC,
Opensearch Leadership Committee

Предисловие

Спасибо за то, что вы приобрели книгу «Поиск на основе искусственного интеллекта»! Эта книга даст вам знания и навыки, необходимые для разработки высокоинтеллектуальных поисковых приложений, которые могут автоматически обучаться на основе каждого обновления контента и взаимодействия с пользователем, постоянно предоставляя все более релевантные результаты поиска.

Нет лучшего времени, чем сейчас, чтобы узнать, как реализовать поиск на основе ИИ. С развитием генеративного ИИ такие методы, как RAG (генерация, дополненная результатами поиска, англ. *Retrieval Augmented Generation*), стали фактическим способом обеспечения систем ИИ актуальными и надежными данными, на основе которых можно получать ответы. Тем не менее «R» в RAG часто является наименее понятным аспектом создания таких систем. Эта книга дает глубокое представление о том, как хорошо выполнять *поиск и извлечение* информации на основе ИИ независимо от того, используете ли вы это для поддержки системы ИИ, создания традиционного поискового приложения или создания новейшего приложения, требующего интеллектуального ранжирования и сопоставления.

За свою карьеру я имел возможность глубоко погрузиться в релевантность поиска, семантический поиск, персонализированный поиск и рекомендации, обработку поведенческих сигналов, семантические графы знаний, обучение ранжированию, LLM и другие базовые модели, поиск по плотным векторам и многие другие возможности поиска на основе ИИ; публиковать исследования в ведущих журналах и на конференциях и, что еще важнее, создавать и поставлять работающее программное обеспечение в больших масштабах. Как основатель Searchkernel, бывший главный специалист по алгоритмам Lucidworks и старший вице-президент по инжинирингу, я также помог поставить многие из этих возможностей сотням самых продвинутых инновационных компаний в мире, чтобы помочь им усилить возможности поиска, которые вы, вероятно, используете каждый день. Я также рад, что Дуг Тернбулл (Reddit, ранее Shopify) и Макс Ирвин (Max.io, ранее OpenSource Connections) также стали соавторами этой книги, опираясь на свой многолетний практический опыт помощи компаниям и клиентам в области поиска и инжиниринга релевантности. В этой книге мы собрали наш многолетний совокупный опыт в практическое руководство, которое поможет вам вывести ваши поисковые приложения на новый уровень. Вы узнаете, как заставить ваши приложения постоянно учиться лучше понимать ваш контент, пользователей и домен, чтобы предоставлять оптимально релевантный опыт при каждом взаимодействии с пользователем.

С наилучшими пожеланиями, когда вы начнете применять поиск на основе ИИ на практике!

—Трей Грейнджер

Благодарности

Прежде всего я хочу поблагодарить мою жену Линдси и моих детей Мелоди, Талли и Оливию. Вы поддерживали меня все эти долгие ночи и выходные, которые я провел за написанием этой книги, и я бы не смог сделать это без вас. Я люблю вас всех!

Далее я хотел бы поблагодарить Дуга Тернбулла и Макса Ирвина за их вклад в эту книгу и в область поиска с использованием искусственного интеллекта (и в информационный поиск в целом). Дуг написал большую часть глав 10–12, а Макс – большую часть глав 13–14 и часть главы 15. Я многому научился у вас обоих и на вашей карьере, и я благодарен за возможность работать с вами над этой книгой.

Далее я хотел бы поблагодарить моего редактора по развитию издательства Manning Марину Майклс. Спасибо за Вашу поддержку и терпение, особенно с учетом того, что сроки выполнения этого масштабного проекта растянулись из-за моей работы в нескольких стартах в ходе производства этого проекта. Качество книги во многом обусловлено Вашим опытом и руководством.

Спасибо также всем остальным сотрудникам Manning, которые работали со мной над разработкой и продвижением проекта: Джону Гатри по технической разработке, Ивану Мартиновичу по ранним выпускам, Майклу Стивенсу по общему видению и направлению и всей маркетинговой команде Manning. Я также благодарю производственную команду Manning за всю их тяжелую работу по форматированию и набору этой книги.

Особая благодарность Гранту Ингерсоллу за написание предисловия. За эти годы я многому научился у Вас, и я очень благодарен за Вашу поддержку.

Далее я хотел бы поблагодарить дополнительных технических авторов книги:

- Дэниела Крауча за его тщательный обзор рукописи книги, его обширный рефакторинг кодовой базы книги и его работу над тем, чтобы сделать книгу в основном не зависящей от конкретных поисковых систем, путем интеграции поддержки plug-and-play для нескольких популярных поисковых систем и векторных баз данных;
- Алекса Отта за его многочисленные технические обзоры книги и за его многочисленные раунды вклада в улучшение кодовой базы книги;
- Мохаммеда Кораема, доктора наук, за его сотрудничество и реализацию алгоритмов обучения графа знаний на основе пользовательских сигналов (глава 6) и персонализированных методов поиска с использованием эмбедингов (глава 9);

- Чао Хан, доктора наук, за его сотрудничество в разработке алгоритмов на основе сигналов для домен-специфичного обнаружения фраз и исправления орфографии.

Я также хотел бы поблагодарить многочисленных читателей, которые предоставили отзывы о ранних версиях этой книги, пока она была в производстве. Ваши отзывы оказали значительное влияние на качество книги. Наконец, я хотел бы поблагодарить рецензентов, которые потратили свое драгоценное время на прочтение рукописи на разных этапах ее разработки и предоставили бесценные отзывы, это Абдул-Басит Хафиз, Адам Дудчак, Эл Кринкер, Ален Кунио, Альфонсо Хесус Флорес Альварадо, Остин Стори, Бхагван Коммади, Дэвид Меза, Давиде Кадамуро, Давиде Фиорентино, Дерек Хэмптон, Гаурав Мохан Тули, Джордж Сейф, Ховард Уолл, Ян Пойнтер, Ишан Курана, Джон Касевич, Кейт Ким, Ким Фальк Йоргенсен, Мария Ана, Марк Джеймс Миллер, Мартин Бир, Мэтт Уэлк, Максим Волгин, Милорад Имбра, Ник Ракочи, Пьерлуиджи Рити, Ричард Воган, Сатей Кумар Саху, Сен Сюй, Шрирам Мачарла, Стив Роджерс, Сумит Пал, Томас Хаук, Тиклу Гангули, Тони Холдройд, Венката Маррапу, Видья Винай, Юдхиеш Равиндранат и Зородзайи Мукуя. Ваши предложения помогли сделать эту книгу лучше.

– Трей Грейнджер

Об этой книге

«Поиск на основе искусственного интеллекта» – пособие по тому, как создавать передовые поисковые системы, которые постоянно обучаются как у ваших пользователей, так и у вашего контента, чтобы обеспечить более предметно-ориентированный и интеллектуальный поиск. Вы изучите современные методы поиска, основанные на науке о данных, такие как:

- семантический поиск с использованием плотных векторных эмбедингов из базовых моделей;
- генерация, дополненная результатами поиска (RAG);
- ответы на вопросы и суммаризация, объединяющие поиск и большие языковые модели (LLM);
- тонкая настройка LLM на основе трансформеров;
- персонализированный поиск на основе пользовательских сигналов и векторных эмбедингов;
- сбор поведенческих сигналов пользователей и построение моделей бустинга сигналов;
- семантические графы знаний для домен-специфичного обучения;
- мультимодальный поиск (гибридные запросы по тексту, изображению, видео и другим типам);
- реализация обобщаемых моделей машинного ранжирования (обучение ранжированию);
- построение кликовых моделей для автоматизации машинного ранжирования;
- методы оптимизации векторного поиска, такие как поиск ANN, квантизация, обучение представлению и би-кодировщики в сравнении с кросс-кодировщиками;
- генеративный поиск, гибридный поиск и передний край поиска.

От современных поисковых систем ожидается, что они будут умными, понимать нюансы запросов на естественном языке, а также предпочтения и контекст каждого пользователя. Эта книга дает вам возможность создавать поисковые системы, которые используют взаимодействие с пользователем и скрытые семантические связи в вашем контенте для автоматического предоставления лучшего, более релевантного результата поиска. Вы даже узнаете, как интегрировать LLM и мультимодальные базовые модели, чтобы значительно ускорить возможности вашей поисковой технологии.

Кому следует прочитать эту книгу

Эта книга предназначена для инженеров поисковых систем, инженеров-программистов и специалистов по данным, которые хотят узнать, как создавать передовые поисковые системы, интегрирующие новейшие методы машинного обучения, чтобы обеспечить более предметно-ориентированный и интеллектуальный поиск. В книге также представлен подробный обзор поиска на основе ИИ для продакт-менеджеров и руководителей предприятий, которые, возможно, не смогут реализовать эти методы самостоятельно, но хотят понять возможности и ограничения поиска на основе ИИ.

Технические читатели, которые хотят извлечь максимальную пользу из этой книги, могут следовать примерам кода Python. Предполагается знакомство с синтаксисом SQL (язык структурированных запросов), поскольку мы решили реализовать многие агрегации данных в этом стандартизированном представлении, когда это возможно. Базовое понимание того, как работают поисковые системы (такие как Elasticsearch, Apache Solr или OpenSearch) или векторные базы данных, также полезно, но не обязательно.

Как организована эта книга: дорожная карта

Книга состоит из 4 разделов, которые включают 15 глав. Часть 1 знакомит с поиском на основе ИИ и современной релевантностью поиска:

- глава 1 содержит обзор поиска на основе ИИ, включая основные понятия и методы, лежащие в основе остальной части книги;
- глава 2 охватывает работу с естественным языком, предоставляя базовые сведения о структуре языка и о том, как он позволяет обучать интеллект на данных;
- глава 3 охватывает основы релевантности поиска, объясняя методы сопоставления и ранжирования, использующие векторные эмбединги и сопоставление ключевых слов;
- глава 4 знакомит с краудсорсинговой релевантностью, охватывая сбор и обработку сигналов взаимодействия с пользователем и предоставляя обзор подходов отраженного интеллекта, которые будут использоваться в последующих главах для автоматической оптимизации алгоритмов релевантности поиска.

Часть 2 охватывает домен-специфичное намерение (intent), уделяя особое внимание использованию контента и взаимодействия с пользователем для оптимизации понимания запросов:

- глава 5 знакомит с изучением графа знаний, уделяя особое внимание извлечению явных графов знаний и неявных семантических графов знаний для детального понимания и расширения запросов;
- глава 6 обучает классификации намерений запросов, устранению неоднозначности различных значений слов и фраз, а также использованию как сигналов контента, так и поведенческих сигналов пользователя для изучения домен-специфичной терминологии.

логии, связанных терминов, опечаток и альтернативных форм терминов;

- глава 7 связывает воедино все, что вы узнали, по мере того как вы создаете конвейер запросов для анализа домен-специфичных намерений, из запросов ваших пользователей для выполнения семантического поиска.

Часть 3 охватывает отраженный интеллект, процесс автоматической оптимизации алгоритмов релевантности поиска на основе текущих взаимодействий с пользователем:

- глава 8 охватывает модели бустинга сигналов, популяризированные модели релевантности, которые автоматически оптимизируют ранжирование ваших самых важных запросов;
- глава 9 охватывает персонализированный поиск, использование совместной фильтрации и плотных векторных эмбедингов для предоставления персонализированных моделей релевантности для каждого пользователя;
- глава 10 представляет ранжирование на основе машинного обучения (также известное как обучение ранжированию), процесс обучения классификатора ранжирования на основе суждений о релевантности пользователя, чтобы действовать в соответствии с обобщенной моделью релевантности;
- глава 11 расширяет обучение ранжированию путем обучения кликовых моделей на входящих поведенческих сигналах пользователя для генерации неявных суждений о релевантности пользователя для непрерывного переобучения модели ранжирования;
- глава 12 представляет активное обучение и A/B-тестирование для преодоления врожденной предвзятости в моделях ранжирования с машинным обучением.

Часть 4 охватывает передовые технологии поиска, описывая новейшие методы и появляющиеся парадигмы в поиске на основе ИИ:

- глава 13 рассказывает о том, как работают трансформеры и LLM, а также о том, как можно использовать и оптимизировать поиск по плотным векторам и приближенный поиск ближайшего соседа (ANN) для встроенных систем, чтобы обеспечить эффективный семантический поиск с использованием би-кодировщиков и кросс-кодировщиков;
- глава 14 демонстрирует тонкую настройку LLM для генерации ответа на вопросы и показывает, как реализовать систему вопрос-ответ, которая извлекает ответы непосредственно из результатов поиска;
- глава 15 завершает наше путешествие, изучая новейшие методы и подходы в поиске на основе ИИ, включая генеративный поиск, генерацию, дополненную результатами поиска (RAG), мультимодальный поиск с использованием базовых моделей, синтетическую генерацию данных и обобщение результатов.

Приложение А предлагает запуск примеров кода, а приложение В охватывает поддерживаемые поисковые системы и векторные базы данных на случай, если вы захотите поменять предпочитаемую вами технологию.

В целом читателям следует очень внимательно прочитать всю часть 1 (особенно главы 1–3), чтобы убедиться, что основные понятия хорошо поняты, прежде чем переходить к другим частям книги. Если вы хотите перейти к следующему разделу, помните о следующих зависимостях между главами:

- глава 1: нет зависимостей;
- глава 2: нет зависимостей;
- глава 3: основана на главе 2;
- глава 4: основана на главах 1, 2, 3;
- глава 5: основана на главах 1, 2, 3;
- глава 6: основана на главах 1, 2, 3, 4, 5;
- глава 7: основана на главах 1, 2, 3, 4, 5, 6;
- глава 8: основана на главах 1, 2, 3, 4;
- глава 9: основана на главах 1, 2, 3, 4, 8;
- глава 10: основана на главах 1, 2, 3;
- глава 11: основана на главах 1, 2, 3, 4, 10;
- глава 12: основана на главах 1, 2, 3, 4, 10, 11;
- глава 13: основана на главах 1, 2, 3;
- глава 14: основана на главах 1, 2, 3, 13;
- глава 15: основана на главах 1, 2, 3, 4, 13, 14.

Мы также настоятельно рекомендуем загрузить блокноты Jupyter и использовать их, чтобы вы могли получить практический опыт работы с данными и примерами кода в книге.

О коде

Хотя методы, описанные в этой книге, широко применимы для использования в большинстве поисковых систем и векторных баз данных, мы решили стандартизировать следующие ключевые технологии для примеров кода:

- *язык программирования* – Python;
- *фреймворк обработки данных* – Spark (PySpark);
- *механизм доставки* – контейнеры Docker;
- *настройка кода и пошаговые инструкции* – блокноты Jupyter;
- *поисковая система / векторная база данных* – Apache Solr (с поддержкой plug-and-play для использования многих других популярных поисковых систем и векторных баз данных).

Запуск примеров кода

Весь код книги написан на Python и поставляется в блокнотах Jupyter, работающих в контейнерах Docker. Это позволяет читателям запускать примеры кода из книги локально в веб-браузере без необходимости дополнительной настройки. Все блокноты разработаны таким

образом, чтобы вы могли запускать их столько раз, сколько захотите, и получать те же результаты, что позволяет вам сосредоточиться на чтении и понимании каждой главы, одновременно выполняя соответствующий код в полностью предварительно настроенной среде.

Инструкции по запуску контейнеров Docker см. в приложении А. Исходный код полностью открыт под лицензией Apache 2.0 и доступен по адресу <https://github.com/treygrainger/ai-powered-search>.

Соглашения по кодированию

С точки зрения соглашений по кодированию мы решили исключить большую часть шаблонного кода из книги, поскольку весь код легко доступен для справки в блокнотах Jupyter. Например, импорты Python обычно исключаются из листингов кода для краткости, если только они не добавляют значительный контекст к примеру кода, например чтобы избежать потенциальной путаницы между пространствами имен.

Аналогично вы обнаружите, что очень длинные листинги кода или выходные данные могут быть сокращены многоточием (...) и что некоторые вспомогательные функции могут иметь свой код, опущенный для экономии места, когда исключенные разделы не важны для понимания смысла примера кода.

В некоторых случаях исходный код был переформатирован; мы добавили переносы строк и переработали отступы до двух пробелов (по сравнению со стандартом Python в четыре пробела), чтобы разместить доступное пространство страницы в книге. В редких случаях даже этого было недостаточно, и листинги включают маркеры продолжения строки (➡). Однако, когда это было возможно, мы использовали продолжения строк Python (\), чтобы сохранить согласованность кодовой базы и листингов книг.

Аннотации кода сопровождают многие листинги, выделяя важные понятия. Кроме того, комментарии в исходном коде обычно удаляются из листингов, когда код описывается в тексте или сопровождающих аннотациях кода.

Полный код для примеров в книге доступен для загрузки с веб-сайта Manning по адресу <https://www.manning.com/books/ai-powered-search>. Самый последний и актуальный репозиторий кода размещен на Github и все еще активно улучшается. Читатели всегда могут загрузить zip-файл последнего кода по этому URL: <https://github.com/treygrainger/ai-powered-search/archive/refs/heads/main.zip>.

Поддержка других поисковых систем и векторных баз данных

Книга также включает поддержку plug-and-play для многих популярных поисковых систем и векторных баз данных, что позволяет вам запускать примеры кода в книге на поисковой системе по вашему выбору. Большая часть книги использует общий движок и абстракцию коллекции, чтобы гарантировать, что код и понятия максимально широко применимы к широкому спектру технологий сопоставления и ранжирования, доступных сегодня. К сожалению, нецелесообразно

включать в книгу дублирующие примеры кода, ориентированные на каждую доступную технологию поисковой системы, поэтому в тех немногих случаях, когда в примерах требуется синтаксис, специфичный для поисковой системы, мы решили стандартизировать поисковую систему Apache Solr с открытым исходным кодом.

Мы используем синтаксис запросов JSON Solr для всех примеров запросов, что делает их легко читаемыми и достаточно простыми для понятийного сопоставления с другими системами. Если вы хотите использовать другую поисковую систему, см. приложение В для получения инструкций о том, как переключаться между различными системами или реализовать свою собственную.

Системные требования для запуска примеров кода

Для комфортного запуска примеров в книге вам понадобится компьютер Mac, Linux или Windows, и мы рекомендуем не менее 8 Гб ОЗУ, чтобы иметь возможность запускать некоторые из наиболее тяжелых задач Spark. Единственная зависимость, которую вам нужно будет установить, – это Docker, после чего вы можете извлечь или построить контейнеры Docker для книги. Контейнеры и наборы данных Docker довольно большие, поэтому мы рекомендуем не менее 25 Гб свободного места на диске для извлечения и обработки всех примеров в книге. Вам может потребоваться изменить системные настройки для Docker, чтобы гарантировать, что у вас выделены эти минимальные объемы памяти и хранилища.

Форум обсуждений liveBook

Покупка данной книги включает бесплатный доступ к liveBook, онлайн-платформе для чтения Manning. Используя эксклюзивные функции обсуждения liveBook, вы можете прикреплять комментарии к книге глобально или к определенным разделам или абзацам. Делать заметки для себя, задавать и отвечать на технические вопросы, а также получать помощь от авторов и других пользователей – это просто. Чтобы получить доступ к форуму, перейдите по ссылке <https://livebook.manning.com/book/ai-powered-search/discussion>. Вы также можете узнать больше о форумах Manning и правилах поведения по ссылке <https://livebook.manning.com/discussion>.

Обязательство Manning перед нашими читателями заключается в предоставлении площадки, где может происходить содержательный диалог между отдельными читателями и между читателями и авторами. Это не обязательство по какому-либо конкретному объему участия со стороны авторов, чей вклад в форум остается добровольным (и неоплачиваемым). Мы предлагаем вам попробовать задать авторам несколько сложных вопросов, чтобы их интерес не рассеивался! Форум и архивы предыдущих обсуждений будут доступны на веб-сайте издателя, пока книга находится в печати.

Другие онлайн-ресурсы

Присоединяйтесь к нашему растущему сообществу! Хотя эта книга служит своевременным и всеобъемлющим руководством по созданию поиска на основе ИИ, мир поиска на основе ИИ продолжает быстро расти и развиваться. Чтобы оставаться в курсе последних и лучших новых методов и технологий поиска на основе ИИ на долгие годы вперед, мы приглашаем вас присоединиться к нашему сообществу *AI-Powered Search Community*.

Присоединившись к сообществу поиска на основе ИИ, вы сможете общаться с тысячами специалистов по поиску и ИИ, включая многих ведущих экспертов (которые уже присоединились) из областей ИИ и поиска информации. Вы можете присоединиться бесплатно по адресу <https://aipoweredsearch.com/community>.

Веб-сайт AI-Powered Search (<https://aipoweredsearch.com>) сосредоточен вокруг трех основных столпов:

- предоставление основного *Руководства* по созданию поиска на основе ИИ (эта книга и последующие обновления);
- предоставление основного *Хаба* для контента поиска на основе ИИ (текущие публикации и обновления контента со всей отрасли);
- создание основного *Сообщества* для поиска на основе ИИ.

Купив эту книгу, вы получили *Руководство*. Веб-сайт AI-Powered Search служит *Хабом* и постоянно обновляемым приложением к этой книге, добавляя новые и появляющиеся методы, к которым вы будете готовы после прочтения этой книги. Теперь мы приглашаем вас *присоединиться к Сообществу*, где вы сможете как делиться тем, над чем работаете, так и взаимодействовать с другими читателями, авторами этой книги и другими ведущими мировыми экспертами по поиску на основе ИИ.

Об авторах



Трей Грейнджер – основатель Searchkernel, компании-разработчика программного обеспечения и консалтинговой компании, которая создает следующее поколение поиска на основе ИИ. Он также является консультантом нескольких стартапов и внештатным профессором компьютерных наук в Университете Фурмана. Ранее он занимал должность технического директора Presearch, децентрализованной поисковой системы в интернете, а также должность главного специалиста по алгоритмам и старшего вице-президента по инжинирингу в Lucidworks, поисковой компании на основе ИИ, чьи поисковые технологии используются сотнями ведущих организаций мира. Он также является соавтором «Solr в действии» (Manning, 2014), ведущей книги по Apache Solr. У Трея более 17 лет опыта в области поиска и науки о данных, включая значительную работу по разработке семантического поиска, персонализации и систем рекомендаций, а также по созданию самообучающихся поисковых платформ, использующих контент и основанный на поведении отраженный интеллект. Результатом этой работы стала публикация десятков исследовательских работ, журнальных статей, презентаций на конференциях и книг на переднем крае интеллектуальных поисковых систем.



Дуг Тернбулл – главный инженер Reddit, бывший инженер по релевантности Spotify и бывший главный технический директор Open-Source Connections. Он является соавтором книги «Релевантный поиск» (Manning, 2016) и написал большую часть глав 10–12 этой книги.



Макс Ирвин – основатель Max.io, специализирующийся на масштабировании моделей ИИ в производстве, и бывший управляющий консультант OpenSource Connections, ведущей консалтинговой компании по релевантности поиска. Макс написал большую часть глав 13–14 и часть главы 15 этой книги.

Об иллюстрации на обложке

Картина на обложке книги «Поиск на основе искусственного интеллекта» – «Homme des Environ's de Rome», или «Человек из окрестностей Рима», взятая из коллекции Жака Грассе де Сен-Совера, опубликованной в 1788 году. Каждая иллюстрация тонко нарисована и раскрашена вручную.

В те дни было легко определить, где жили люди, какова была их профессия или положение в жизни, просто по их одежде. Мэннинг прославляет изобретательность и инициативу компьютерного бизнеса с помощью обложек книг, основанных на богатом разнообразии региональной культуры многовековой давности, оживленных фотографиями из таких коллекций, как эта.

Часть 1

Современная релевантность поиска

Поисковые системы служат шлюзом для доступа к большей части человеческих знаний. Поисковые системы (поисковые движки) предлагают запрашиваемый кеш интернета, позволяющий мгновенно находить информацию по любой теме на миллиардах веб-сайтов. Генеративный ИИ (искусственный интеллект) в значительной степени опирается на поисковые системы для выполнения генерации, дополненной результатами поиска (RAG), представляющей собой процесс для поиска релевантного контекста, который предоставляется моделям ИИ, чтобы они генерировали точные ответы на входящие запросы.

Алгоритмы поиска также обеспечивают сопоставление и ранжирование в большинстве приложений, управляемых данными: от электронной коммерции до электронной почты, социальных сетей, корпоративных интрасетей и частных файловых систем. Для хорошего выполнения поиска требуется оптимизация релевантности поиска – способность находить и ранжировать наиболее релевантные результаты для заданного запроса.

В этой первой части книги мы рассмотрим современную релевантность поиска. Глава 1 представляет собой введение в поиск на основе ИИ, где выделяются основные темы, которые вы изучите на протяжении всей книги. Глава 2 посвящена работе с естественным языком

и предоставляет справочную информацию о ключевых понятиях обработки естественного языка (NLP), необходимых для реализации поиска на основе ИИ. Затем глава 3 углубляется в ранжирование по релевантности, охватывая механизмы того, как поисковые системы и векторные базы данных находят и ранжируют лучшие результаты для заданного запроса. Наконец, в главе 4 рассматривается краудсорсинговая релевантность – процесс использования постоянного взаимодействия пользователей с результатами поиска для изучения моделей, которые повышают рейтинг релевантности.

1

Знакомство с поиском на основе ИИ

https://t.me/it_boooks/2

В этой главе рассматривается:

- что такое поиск на основе ИИ?
- понимание намерения пользователя;
- как работает поиск на основе ИИ;
- контент и поведенческий интеллект;
- разработка поисковой системы на основе ИИ.

Поле поиска стало пользовательским интерфейсом по умолчанию для взаимодействия с данными в большинстве современных приложений. Если вы подумаете о каждом крупном приложении или веб-сайте, которые вы используете ежедневно, одно из первых действий, которое вы, вероятно, сделаете при каждом посещении, – это введете запрос в это поле, чтобы найти контент или действия, наиболее релевантные для вас.

Когда вы не занимаетесь поиском явно, вы можете вместо этого потреблять потоки контента, настроенные в соответствии с вашими вкусами и интересами. Будь то видеорекомендации, продукты для покупки, приоритетные электронные письма, новостные статьи или другой контент, вы, вероятно, смотрите на отфильтрованные или ранжиро-

ванные результаты и получаете возможность либо пролистать, либо явно отфильтровать контент с помощью собственного запроса.

Для большинства людей фраза «поисковая система» (или «поисковый движок») вызывает мысли о веб-сайтах вроде Google, Bing или Baidu, которые позволяют выполнять запросы на основе сканирования всего общедоступного интернета. Однако реальность такова, что поиск теперь доступен практически во всех наших цифровых взаимодействиях каждый день на многочисленных веб-сайтах и в приложениях, которые мы используем.

Эти поисковые движки далеки от статичности. Мы видим коммерческие технологии, такие как ChatGPT от OpenAI, Claude от Anthropic и Gemini от Google, а также сотни других более открытых больших языковых моделей (LLM), таких как Llama от Meta и Mixtral от Mistral, с исходным кодом и весами моделей, опубликованными для публичного использования. Все они служат моделями мировой информации, которые могут генерировать интерпретации и ответы на произвольные запросы. Эти модели активно интегрируются в основные поисковые системы и продолжают оказывать сильное влияние на эволюцию поиска на основе ИИ.

Хотя ожидаемым ответом на запрос в поисковой строке исторически могло быть возвращение «десяти синих ссылок» – списка ранжированных документов, которые пользователь может изучить далее, чтобы найти информацию в ответ на свой запрос, – ожидания относительно уровня интеллекта поисковых технологий в последние годы резко возросли.

Сегодня пользователи ожидают, что поисковая технология будет:

- *домен-осведомленной* – поисковая технология должна понимать сущности, терминологию, категории и атрибуты каждого конкретного варианта использования и корпуса документов, а не просто использовать общую статистику по строкам текста;
- *контекстуальной и персонализированной* – поисковая технология должна иметь возможность учитывать контекст пользователя (локация, последний поиск, профиль, предыдущие взаимодействия, рекомендации пользователя и классификация пользователя), контекст запроса (другие ключевые слова, похожие поиски) и контекст предметной области (инвентарь, бизнес-правила, терминология, специфичная для предметной области), чтобы лучше интерпретировать намерения пользователя;
- *разговорной* – поисковая технология должна иметь возможность взаимодействовать на естественном языке и направлять пользователей через многоэтапный процесс обнаружения, изучая и запоминая соответствующую новую информацию по ходу дела;
- *мультимодальной* – поисковая технология должна иметь возможность разрешать запросы, выдаваемые текстом, голосом, изображениями, видео или другими типами контента, и использовать эти запросы для поиска по другим типам контента;

- *интеллектуальной* – поисковая технология должна иметь возможность предоставлять предиктивный ввод и понимать, что имеют в виду пользователи (исправление орфографии, обнаружение фраз и атрибутов, классификация намерений, понятийный поиск), чтобы предоставлять правильные ответы в нужное время и постоянно становиться умнее;
- *вспомогательной* – она должна выходить за рамки предоставления только ссылок и предоставлять ответы, резюме, объяснения и доступные действия.

Многие из этих возможностей обеспечиваются LLM, в то время как другие управляются путем анализа поведения пользователя и создания домен-специфичных профилей персонализации, графов знаний и моделей ранжирования.

Интерфейсы поиска также развиваются, чтобы включать больше сеансов чат-ботов и разговорного поиска информации, поскольку LLM становятся все более повсеместными, но даже лучшие сегодняшние модели борются с галлюцинациями¹ и сбиваются с рельсов, если не привязаны к фактическому источнику информации, такому как индекс поисковой системы, чтобы надежно находить и возвращать информацию из надежных источников. *Генерация, дополненная результатами поиска (RAG)*, метод использования поисковой системы или векторной базы данных в качестве источника знаний для предоставления LLM точной и актуальной информации в качестве контекста, является одним из самых надежных методов повышения точности генеративных моделей ИИ на сегодняшний день.

Цель поиска на основе ИИ заключается в использовании автоматизированных методов машинного обучения для предоставления всех этих желаемых возможностей. В то время как многие организации

¹ Галлюцинации в программировании – это генерируемые искусственным интеллектом результаты, которые не имеют под собой реальной основы. Некоторые проявления галлюцинаций в программировании:

- заведомо ложная информация. ИИ может давать определение несуществующим явлениям, предлагать списки фиктивных научных исследований, генерировать биографии людей, которых в действительности никогда не было;
- неправильная интерпретация данных. Иногда ИИ неверно понимает информацию и делает ошибочные выводы. Например, нейросеть анализирует поток автомобилей и выявляет те, где водитель не пристегнут. В ее выборку периодически попадают не только нарушители, но и владельцы праворульных машин;
- неуникальный результат. Периодически программы лишь слегка видоизменяют готовый контент, а не создают новый. Например, они предлагают в качестве сгенерированной картинки фотографию реального человека;
- выполнение задач, которые никто не ставил. В некоторых случаях нейросеть начинает делать то, о чем ее не просили. Например, не просто переводит заданную часть текста, но и дописывает его.

Причины галлюцинаций могут быть разными, например обучение систем ИИ на неточных или недостаточных данных. Возможны и недочеты в алгоритмах, которые заставляют ИИ обобщать данные неправильно и учитывать фейковую информацию. Чтобы распознать галлюцинацию, любой ответ нейросети нужно проверять по общим правилам фактчекинга. – *Прим. ред.*

начинают с простого текстового поиска и тратят много лет, пытаясь вручную оптимизировать списки синонимов, бизнес-правила, онтологии, веса полей и бесчисленное множество других аспектов своей конфигурации поиска, некоторые начинают понимать, что большую часть этого процесса можно автоматизировать.

В ходе прочтения книги вы научитесь применять множество ключевых методов поиска на основе искусственного интеллекта, таких как:

- использование LLM для интерпретации запросов, эмбединга, генерации ответов на вопросы и обобщения результатов;
- тонкая настройка LLM для поиска и извлечения ответов на вопросы;
- сбор и использование пользовательских сигналов для краудсорсинговой релевантности;
- модели бустинга сигналов;
- обучение графа знаний как на основе сигналов, так и на основе контента;
- семантические графы знаний;
- классификация намерений запросов и устранение неоднозначности смысла запросов;
- персонализированный поиск и рекомендации;
- машинное ранжирование (обучение ранжированию);
- модели кликов для неявной обратной связи по релевантности;
- избежание предвзятости в моделях ранжирования с помощью активного обучения;
- гибридный поиск и мультимодальный поиск по тексту, изображениям и смешанным типам контента;
- семантический поиск с использованием как графов знаний, так и LLM.

Эта книга представляет собой руководство на основе примеров по наиболее применимым алгоритмам и методам машинного обучения, которые обычно используются для создания интеллектуальных поисковых систем. Мы не только рассмотрим ключевые понятия, но и предоставим примеры кода для повторного использования, охватывающие методы сбора и обработки данных, а также стратегии интерпретации и релевантности запросов с самообучением, используемые для предоставления возможностей поиска на основе ИИ в ведущих современных организациях, – надеемся, что скоро вы тоже сможете воспользоваться ими!

1.1. Что такое поиск на основе ИИ?

До ноября 2022 года, когда OpenAI выпустила ChatGPT в мир как обобщенный алгоритм, с которым нетехнические пользователи могли общаться для решения многих задач, определение искусственного интеллекта было немного туманным для широкой публики. Считалось, что он включает в себя такие вещи, как беспилотные автомоби-

ли, автономные роботы и другие футуристические технологии, благодаря которым компьютеры кажутся разумными, но многим ИИ казался скорее модным маркетинговым словом, чем четко определенным термином. Однако в индустрии программного обеспечения уже много лет существует более конкретное определение.

В контексте разработки программного обеспечения термин *искусственный интеллект* обычно описывает любую компьютерную программу, которая может выполнять задачу, ранее требующую человеческого интеллекта. Эта программа часто включает в себя методы машинного обучения, что позволяет ей учиться на данных и со временем улучшать свою производительность. При этом даже системы на основе правил, которые не используют методы машинного обучения, но генерируют обратную связь, подобную человеческой, также традиционно считаются системами «ИИ». Мы примем это более общее определение ИИ в этой книге, хотя в первую очередь мы обсудим аспекты машинного обучения ИИ.

Термин *поиск* (*поисковый движок* или *поисковая система*) также рассматривается широкой публикой как относящийся к поисковым системам в интернете, таким как Google или Bing. В разработке программного обеспечения этот термин также используется для описания любой технологии, которая позволяет пользователям запрашивать и находить информацию. Поиск обычно включает в себя как минимум два важных шага – поиск документов, соответствующих запросу (*сопоставление*), а затем упорядочивание этих документов по релевантности запросу (*ранжирование*). Поиск также может включать в себя множество шагов предварительной обработки для лучшего понимания запроса и шагов постобработки для извлечения ответов или суммирования результатов из сопоставленных документов. Поиск часто является основным способом, которым пользователи находят информацию, независимо от того, выполняют ли они общий поиск в интернете, поиск по продуктам, корпоративный поиск, поиск видео, изображений или любой из сотен других распространенных вариантов использования и ранжирования информации. Это также основной способ, с помощью которого генеративные системы ИИ быстро находят обновленный фактический контент для использования в качестве контекста для своих подсказок.

Но что такое поиск на основе ИИ и чем он отличается от традиционного «поиска»? Многие модные слова, такие как «ИИ», «машинное обучение», «наука о данных» и «глубокое обучение», часто используются как взаимозаменяемые, и важно понимать различия и то, как они пересекаются с поиском на основе ИИ. Рисунок 1.1 демонстрирует важные отношения между этими связанными областями.

Машинное обучение – это подмножество ИИ, которое фокусируется на использовании данных для обучения моделей для выполнения задач на основе знаний, полученных из обучающих данных. Глубокое обучение – это еще одно подмножество машинного обучения, фоку-

сирующееся на обучении искусственных нейронных сетей – алгоритмов, которые частично имитируют структуру человеческого мозга, – чтобы научиться решать сложные задачи. На рис. 1.1 обратите внимание, что глубокое обучение – это полностью замкнутое подмножество машинного обучения, которое затем является полностью замкнутым подмножеством искусственного интеллекта. Наука о данных – это дисциплина, которая во многом пересекается с ИИ и поиском, но также содержит другие отдельные области фокусировки, поэтому она не является полностью надмножеством или подмножеством того и другого.



Рис. 1.1. Поиск на основе ИИ включает все технологии и методы на пересечении областей поиска и ИИ. Они во многом пересекаются и используют области науки о данных, машинного обучения и глубокого обучения

В этой книге мы уделяем особое внимание пересечению поиска (также известного как *информационный поиск*) и ИИ и, в частности, применению методов машинного обучения и глубокого обучения для повышения релевантности результатов поиска и автоматизации процесса настройки релевантности поиска. Создание поиска на основе ИИ включает в себя множество известных методов машинного обучения, но также и множество методов, специфичных для информационного поиска и области поиска. На рис. 1.2 представлен категоризированный список некоторых ключевых методов поиска на основе ИИ, которые мы рассмотрим в этой книге, разбитый по тому, являются ли они методами глубокого обучения, другими методами машинного обучения, не требующими глубокого обучения, или другими методами искусственного интеллекта, не требующими машинного обучения.

Если говорить только о категории ИИ, то системы ответов на вопросы, виртуальные помощники, чат-боты и релевантность на основе правил – все это примеры методов ИИ, которые часто строятся с использованием машинного обучения, но *не требуют* машинного обучения. Многие создали чат-ботов, полностью основанных на правилах, чтобы понимать различные высказывания и намерения пользователя, и аналогичным образом системы ответов на вопросы могут быть построены исключительно на правилах и онтологиях. Тем не менее машинное обучение часто используется для изучения этих видов правил и онтологий, поэтому границы между этими категориями часто размыты.



Рис. 1.2. Специфические методы поиска на основе ИИ, разбитые на методы глубокого обучения, другие методы машинного обучения, не требующие глубокого обучения, или другие методы искусственного интеллекта, не требующие машинного обучения

Когда алгоритмы начинают использовать данные для обучения моделей, мы вступаем в подкатегорию машинного обучения поиска на основе ИИ. Мы используем поведенческие сигналы от пользователей поисковой системы (клики, лайки, добавления в корзину, покупки и т. д.) для построения моделей, которые могут научиться лучше ранжировать документы. Это может включать в себя модель бустинга сигналов (топовые документы по запросу или категории), модели совместной фильтрации, которые генерируют рекомендации или персонализируют результаты поиска, и классификаторы ранжирования (обучение ранжированию), которые обучаются на основе контента и поведенческих сигналов для лучшего ранжирования результатов. Машинное обучение также используется для изучения графов знаний, представляющих собой графы сущностей, понятий и их отношений, которые можно использовать для лучшего понимания домена и лучшей интерпретации пользовательских запросов. Семантический поиск (поиск по смыслу, а не только по ключевым словам) может быть включен с помощью таких графов знаний наряду с традиционными подходами к обработке естественного языка, классификацией намерений запросов, кластеризацией документов и другими методами, управляемыми запросами пользователей, документами и поведенческими сигналами пользователей. Наконец, в подкатегории глубокого обучения поиска на основе ИИ мы видим использование нейронных сетей для создания моделей, которые могут понимать запросы пользователей и документы, а также ранжировать и суммировать результаты поиска. Здесь текст используется для обучения LLM понимать

значение слов и фраз, генерировать ответы на вопросы и генерировать резюме документов. LLM – это тип базовой модели, которая может интерпретировать текстовый контент и часто обучается на огромных объемах текста из интернета. Базовые модели также могут обучаться на других типах контента, помимо текста (изображения, аудио, видео), чтобы обеспечить мультимодальный поиск по этим типам контента: поиск текста в изображении, текста в аудио, изображения в видео и т. д. LLM также используются для создания эмбеддингов¹, которые являются векторными представлениями контента, представляющими смысл контента. Поскольку основная задача поисковой системы – находить и ранжировать контент, похожий на входящий запрос, эти эмбеддинги обеспечивают сложную возможность поиска по смыслу запроса и значительно улучшают понимание и ранжирование запроса. Дальнейшая настройка базовых моделей на конкретные цели или наборы данных, специфичные для определенной области, также значительно улучшит их понимание нюансов этих областей или вариантов использования.

Базовые модели сжимают большой объем человеческих знаний (часто большую часть интернета), предоставляя им широкое понимание в большинстве областей. Однако это сжатие знаний является сжатием с потерями – исходные данные не сохраняются, а конкретные факты и понятия можно легко спутать. Известно, что базовые модели галлюцинируют ответы на вопросы, что делает их в целом ненадежными для генерации ответов на фактические вопросы. В результате в дополнение к поисковым системам, использующим базовые модели для улучшения понимания и ранжирования запросов, мы также видим, что они активно используются для RAG – где поиск служит источником знаний, на который базовые модели могут полагаться для получения точной и актуальной информации в качестве контекста для генеративных задач ИИ. Мы подробно рассмотрим каждый из этих методов поиска на основе ИИ в этой книге. Но сначала давайте обсудим цели поиска на основе ИИ и то, чем он отличается от традиционного поиска.

1.2. Что такое намерения пользователя

Чтобы обеспечить поиск на основе ИИ, нам понадобится целостное понимание измерений, задействованных в интерпретации намерения пользователя² и возврате контента, соответствующего этому намере-

¹ Эмбеддинг, «векторное вложение» или «вложение», вектор эмбеддинга – это числовое представление данных, которое фиксирует определенные особенности. Это математическое представление объектов в непрерывном векторном пространстве, используемое для захвата семантического смысла или свойств этих объектов. На практике векторные эмбеддинги – это массивы реальных чисел фиксированной длины (обычно от сотен до тысяч элементов), генерируемые моделями машинного обучения. С помощью векторных эмбеддингов модель находит сходство между различными словами, фокусируясь на значимой информации о каждом из них. – *Прим. ред.*

² Намерение, англ. *intent*, – это сообщение, которое указывает системе, что нужно сделать. – *Прим. ред.*

нию. В области поиска информации поисковые системы и рекомендательные системы являются двумя наиболее популярными технологиями, используемыми для предоставления соответствующего контента, необходимого для удовлетворения информационных потребностей пользователей. Многие организации рассматривают поисковые движки и рекомендательные системы как отдельные технологии, решающие различные варианты использования. Обычно разные команды в одной организации – часто с разными наборами навыков – работают независимо над отдельными поисковыми и рекомендательными движками. В этом разделе мы обсудим, почему разделение поиска и рекомендаций на независимые функции и команды часто может приводить к неидеальным результатам.

1.2.1. Что такое поисковая система?

Поисковая система (поисковый движок) обычно рассматривается как технология для явного ввода запросов и получения ответа (рис. 1.3). Обычно она предоставляется конечным пользователям через текстовое поле, в которое пользователь может вводить ключевые слова или вопросы. Результаты часто возвращаются в виде списка вместе с дополнительными параметрами фильтрации, которые позволяют дополнительно уточнить исходный запрос. Используя этот механизм, поиск используется как инструмент для прямого обнаружения соответствующего контента. Когда пользователь завершает сеанс поиска, он обычно может ввести новый запрос и начать с чистого листа, игнорируя контекст предыдущих поисков.

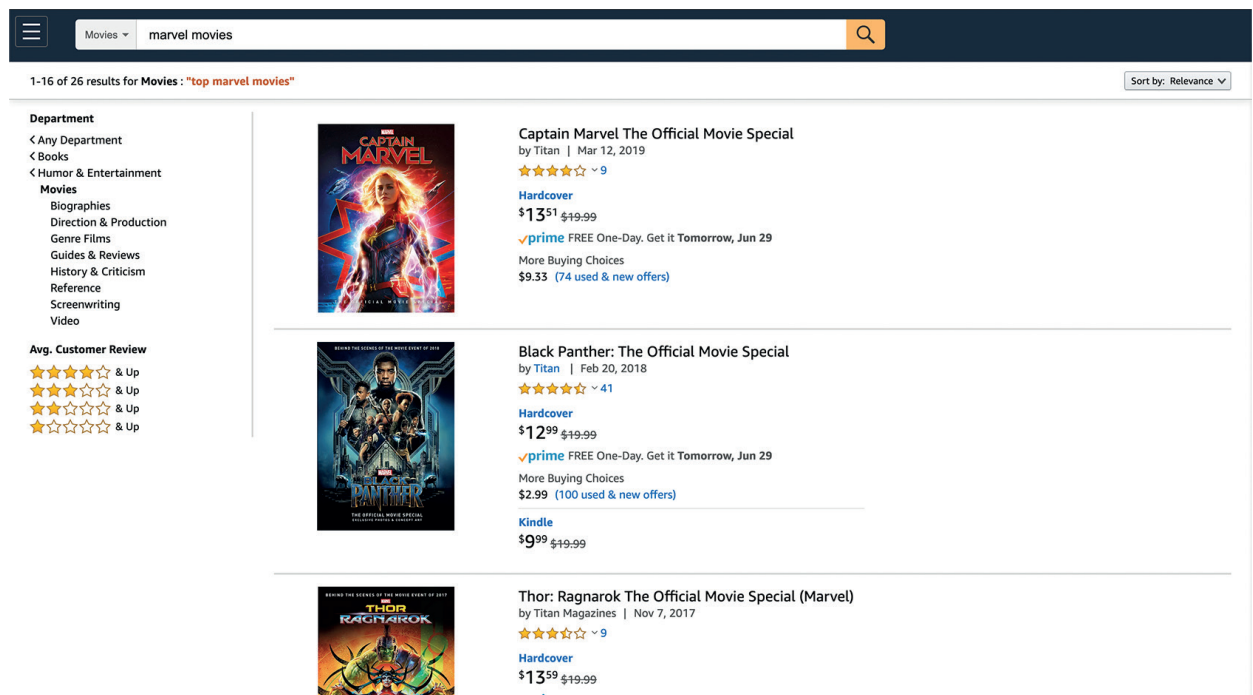


Рис. 1.3. Типичный опыт поиска, когда пользователь вводит запрос и видит результаты поиска с параметрами фильтрации для поддержки дальнейшего уточнения результатов поиска

Поисковая система является одним из наиболее кросс-функциональных видов систем в мире разработки программного обеспечения. Большинство базовых технологий поисковых систем разработаны для работы в масштабируемом режиме, обслуживая большие объемы запросов по миллионам, миллиардам или даже триллионам документов и предоставляя результаты за сотни миллисекунд или меньше. Во многих случаях требуется обработка в реальном времени и поиск в режиме, близком к реальному времени, по вновь полученным данным, и все это должно быть распараллелено на нескольких серверах для масштабирования и соответствия таким высоким требованиям к производительности.

Реализация поисковых систем также требует значительной работы по созданию структур данных, специфичных для поиска, таких как инвертированный индекс или хранилище векторов на основе ИНС, понимания линейной алгебры и оценки сходства векторов, опыта анализа текста и обработки естественного языка, а также знания многочисленных типов моделей данных и возможностей, специфичных для поиска (проверка орфографии, автоподсказка, фасетирование, выделение текста, эмбединг и т. д.).

Чтобы поисковая система полностью интерпретировала намерения пользователя, крайне важно, чтобы вы объединили глубокое понимание вашего контента, ваших пользователей и вашего домена. Мы вернемся к тому, почему это важно, после краткого обсуждения связанной темы рекомендательных систем.

1.2.2. Что предлагают рекомендательные системы

Большинство людей думают о рекомендательных системах (или «рекомендательных движках») как о системах, которые не принимают прямой ввод пользователя, а вместо этого предоставляют контент на основе того, что система узнает о них, вычисляя наилучшие соответствия их интересам и поведению. Эти интересы определяются различными способами через предпочтения пользователя, поведение пользователя, просмотренный контент и т. д. Такое отсутствие прямого пользовательского ввода для рекомендательных систем находится в прямом противоречии с поисковыми системами, которые традиционно считаются технологией, требующей явных запросов, инициируемых пользователем.

Если вы регулярно посещаете Amazon.com или любой другой крупный сайт электронной коммерции, вы, несомненно, знакомы с разделами рекомендательных систем, в которых говорится, что «исходя из вашего интереса к этому продукту, вам также может понравиться...», или просто рекомендуется список продуктов на основе вашей коллективной истории просмотров и покупок, как в примере на рис. 1.4. Эти рекомендации часто приносят компаниям значительный доход и помогают клиентам находить релевантный, персонализированный и связанный контент, который часто явно дополняет то, что они ищут.

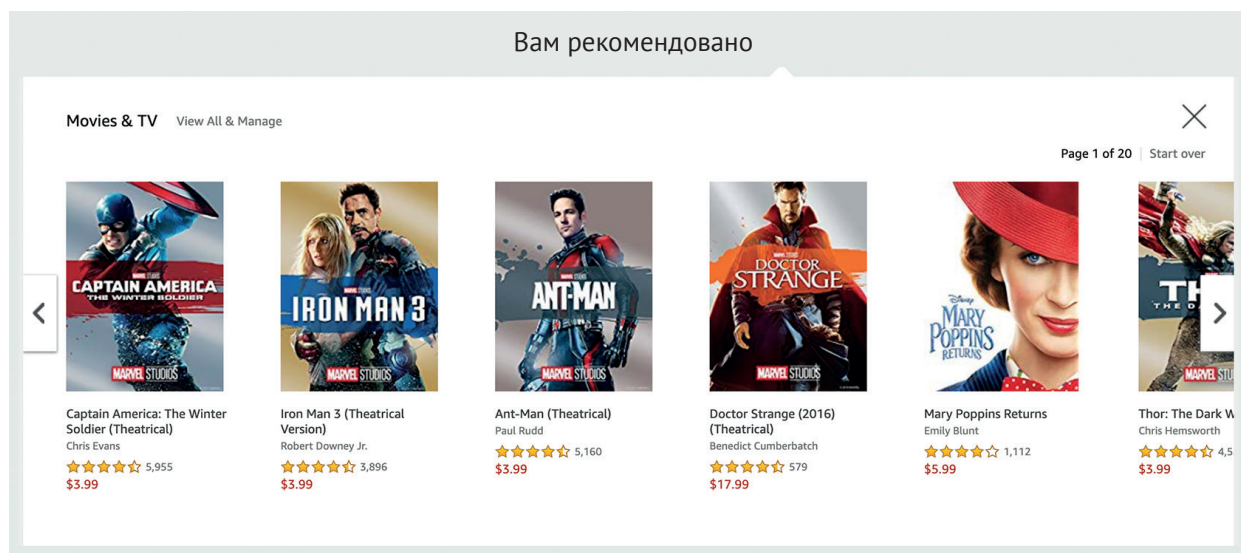


Рис. 1.4. Рекомендации, основанные на проявленном пользователем интересе к похожим продуктам

Алгоритмы рекомендаций можно условно разделить на три категории:

- *рекомендательные системы на основе контента* – сопоставление на основе атрибутов продуктов или пользователей;
- *рекомендательные системы на основе поведения* – сопоставление на основе соответствия взаимодействий похожих пользователей с похожими продуктами;
- *мультимодальные рекомендательные системы* – выполняют гибридное сопоставление на основе как схожих атрибутов контента, так и перекрывающихся взаимодействий на основе поведения.

1.2.3. Спектр персонализации между поиском и рекомендациями

Ключевое различие между поисковыми системами и рекомендательными системами заключается в том, что поисковые системы обычно управляются пользователем и ориентируются на явно введенные запросы пользователей, тогда как рекомендательные системы обычно не принимают прямого ввода пользователя и вместо этого рекомендуют – на основе уже известных или предполагаемых знаний – то, что пользователь может захотеть увидеть дальше.

На самом деле эти две системы являются двумя сторонами одной медали, и рассмотрение их как отдельных систем создает ложную дихотомию. Цель в обоих случаях – понять, что ищет пользователь, и предоставить релевантные результаты для удовлетворения информационных потребностей этого пользователя. Широкий спектр возможностей персонализации лежит в пределах спектра между поисковыми и рекомендательными системами.

Предполагая, что у вас есть как явные запросы, так и профиль персонализации, специфичный для пользователя, при попытке найти контент для ваших конечных пользователей, вы можете сделать любое из следующего:

- *традиционный поиск по ключевым словам* – игнорировать профиль и использовать только явные вводы;
- *персонализированный поиск* – использовать профиль неявно вместе с другим явным вводом пользователя;
- *рекомендации, управляемые пользователем*, – явно используйте профиль и предоставьте пользователю возможность его настройки;
- *традиционные рекомендации* – явно используйте профиль без возможности его настройки пользователем.

На рис. 1.5 показан этот спектр персонализации.

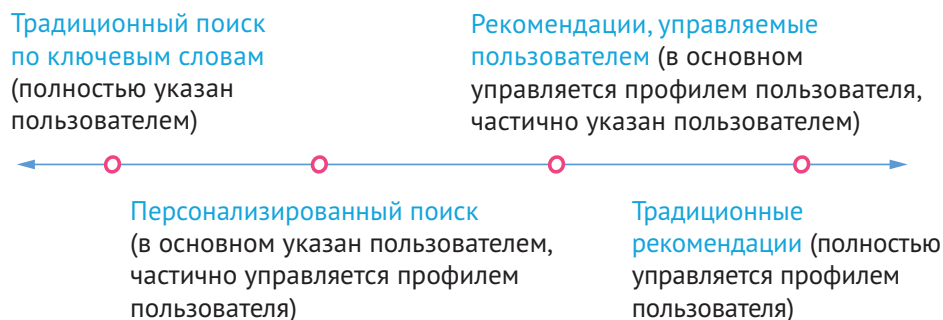


Рис. 1.5. Спектр персонализации, показывающий традиционный поиск по ключевым словам и традиционные рекомендации как два конца широкого континуума

Хотя два конца этого спектра персонализации представляют собой крайности, они также являются двумя наиболее распространенными подходами. К сожалению, одна из самых больших ошибок, которую мы видим во многих организациях, заключается в том, что команды построены вокруг убеждения, что поиск и рекомендации – это отдельные задачи. Это часто приводит к тому, что команды по науке о данных создают сложные модели персонализации и сегментации, способные только на рекомендации, а не на поиск, а инженерные команды создают крупномасштабные механизмы сопоставления ключевых слов, которые не могут легко воспользоваться преимуществами надежных моделей, созданных командами по рекомендациям.

Чаще всего команды по рекомендациям укомплектованы специалистами по данным с минимальным опытом поиска информации, а команды по поиску часто укомплектованы инженерами с минимальным опытом в области науки о данных. Из-за закона Конвея («организации, которые проектируют системы... ограничены в создании проектов, которые являются копиями коммуникационных структур этих организаций») это в конечном итоге приводит к проблемам решения задач по всему спектру персонализации (особенно в середине), которые требуют наилучшего от обеих команд. В этой книге мы фокусируемся на общих методах, которые позволяют сделать поиск более интеллектуальным, а рекомендации – более гибкими за счет единого подхода. Поисковые платформы на базе ИИ должны иметь возможность постоянно учиться как у ваших пользователей, так и у вашего контента, а затем позволять вашим пользователям направлять результаты, чтобы они продолжали улучшаться.

1.2.4. Семантический поиск и графы знаний

Мы представили поиск и рекомендации как спектр персонализации на рис. 1.5 с персонализированным поиском и рекомендациями, управляемыми пользователем, между ними, но есть еще одно измерение, которое имеет решающее значение для создания хорошей поисковой системы на базе ИИ, – глубокое понимание заданной области. Недостаточно сопоставлять ключевые слова и рекомендовать контент на основе того, как пользователи коллективно взаимодействуют с документами. Система также должна узнать как можно больше о домене. Это включает:

- изучение всех важных фраз, синонимов и связанных терминов, относящихся к предметной области;
- определение сущностей в документах и запросах;
- создание графа знаний, связывающего эти сущности;
- устранение неоднозначности множества нюансов значений, представленных терминологией, относящейся к предметной области;
- возможность эффективно анализировать, интерпретировать и понятийно сопоставлять нюансы намерений пользователей в вашем домене.

На рис. 1.6 показан пример семантического анализа запроса, целью которого является поиск «вещей» (известных сущностей) вместо «строк» (просто сопоставления текста).

Цель: поиск по «вещам», а не по «строкам»...

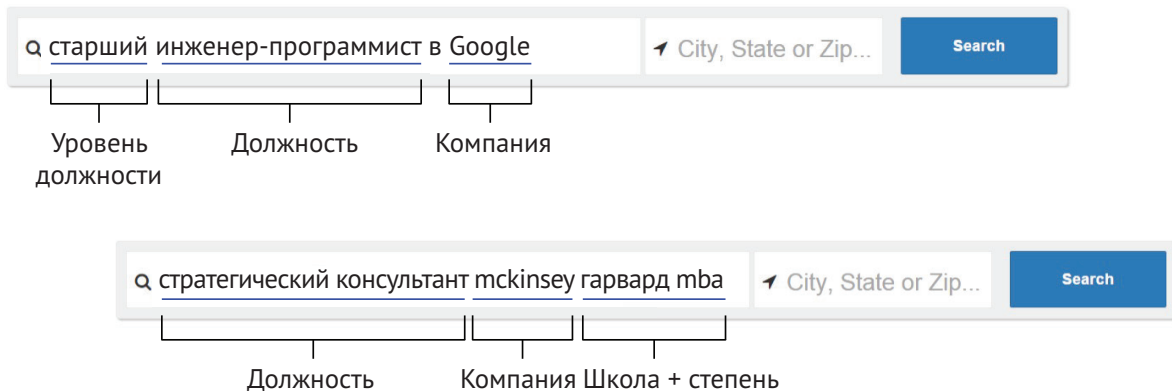


Рис. 1.6. Семантический анализ запроса, демонстрирующий понимание сущностей («вещей»), представленных терминами запроса

Чтобы сделать свои поиски более интеллектуальными, многие компании тратят значительные деньги, нанимая большие команды для ручного создания словарей и графов знаний для определения связей между сущностями в запросах своих пользователей. В этой книге основное внимание уделяется более масштабируемому подходу: созданию поисковой системы на базе ИИ, которая может автоматически изучать эти связи непрерывно. Мы также рассмотрим дополнительные методы семантического поиска, включая поиск по плотным векторам, поиск по эмбедингам и генеративный поиск с использованием LLM.

1.2.5. Что такое измерения намерения пользователя

Мы обсудили важные роли традиционного поиска по ключевым словам, рекомендаций и спектра персонализации между ними. Мы также обсудили необходимость семантического поиска для обеспечения понимания вашего контента и запросов ваших пользователей в зависимости от предметной области. Все это ключевые столпы единой, более масштабной цели: полного понимания намерения пользователя. Рисунок 1.7 демонстрирует взаимодействие между каждым из этих ключевых столпов намерения пользователя.



Рис. 1.7. Измерения намерения пользователя: сочетание понимания контента, понимания пользователя и понимания домена

Верхний левый круг на рис. 1.7 представляет *понимание контента* – способность находить нужный контент на основе ключевых слов, языковых шаблонов и известных соответствий атрибутов. Верхний правый круг представляет *понимание пользователя* – способность понимать особые предпочтения каждого пользователя и использовать их для возврата более персонализированных результатов. Наконец, нижний круг представляет *понимание домена* – способность интерпретировать слова, фразы, концепты¹, сущности и тонкие интерпретации и отношения между каждым из них в вашем собственном домен-специфичном контексте.

Запрос только в круге понимания контента представляет традиционный *поиск по ключевым словам*, позволяющий сопоставлять ключевые слова, но без использования какого-либо домен-специфичного контекста или пользователя. Запрос только в круге понимания поль-

¹ Концепт в лингвистике – это структурно-содержательная единица, отражающая совокупность знаний, представлений, мнений об объекте. – *Прим. ред.*

зователя будет рекомендациями из совместной фильтрации без возможности для пользователя переопределять вводимые данные и без понимания домена или содержания базовых документов. Запрос только в круге понимания домена может быть структурированным запросом по известным тегам, категориям или сущностям или даже интерфейсом, похожим на просмотр, который позволяет исследовать *графы знаний* этих домен-специфичных сущностей и их отношений, но без какой-либо специфичной для пользователя персонализации или возможности находить произвольные термины, фразы и содержание.

Когда традиционный поиск по ключевым словам и рекомендации пересекаются, мы получаем *персонализированный поиск* или управляемые рекомендации. Когда традиционный поиск по ключевым словам и графы знаний пересекаются, мы получаем *семантический поиск*: интеллектуальный поиск, ориентированный на домен. Наконец, когда рекомендации и графы знаний пересекаются, мы получаем более интеллектуальные рекомендации с учетом домена, которые соответствуют краудсорсинговым взаимодействиям пользователей в похожих документах, а также пониманию важных атрибутов этих документов с учетом домена.

Святой Грааль для поиска на основе ИИ – использовать пересечение всех трех категорий: семантического поиска, персонализированного поиска и рекомендаций с учетом домена. То есть, чтобы по-настоящему понять намерение пользователя, нам необходимо все нижеперечисленное:

- экспертное понимание домена, в котором пользователь делает поиск;
- экспертное понимание пользователя и его предпочтений;
- экспертная способность сопоставлять и ранжировать произвольные запросы по любому контенту.

Поиск на основе ИИ начинается с трех столпов намерения пользователя (контент, домен и пользователь), а затем использует интеллектуальные алгоритмы для постоянного обучения и совершенствования в каждой из этих областей. Это обучение включает такие методы, как автоматическое обучение критериям ранжирования, автоматическое обучение предпочтениям пользователя и автоматическое обучение графам знаний и языковым моделям представленного домена. В конце концов сбалансированное сочетание этих трех подходов дает ключ к оптимальному пониманию пользователей и их намерений запроса, что является конечной целью нашей поисковой системы на основе ИИ.

1.3. Как работает поиск на основе ИИ?

Мы изложили нашу конечную цель сопоставления намерения пользователя с помощью понимания контента, понимания пользователя и понимания домена. С учетом этого фона давайте завершим эту главу обзором фактических компонентов, необходимых для предоставления поисковой платформы на основе ИИ. Поисковый интеллект обычно развива-

ется по предсказуемой итеративной схеме с течением времени, как показано на рис. 1.8. Базовый поиск по ключевым словам является типичной отправной точкой для организаций. После запуска в производство они понимают, что релевантность поиска необходимо улучшить, и начинают вручную настраивать веса полей, делать бустинг¹, анализ текста и языка, а также вводить дополнительные функции и возможности.

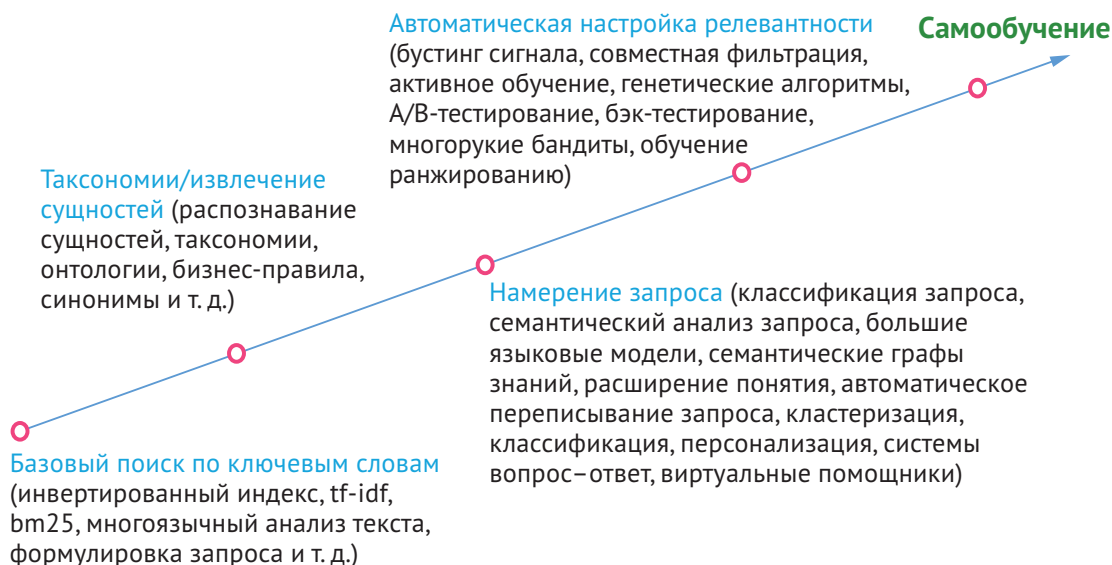


Рис. 1.8. Типичная прогрессия поисковой разведки, от базового поиска по ключевым словам до полностью самообучающейся поисковой платформы

В конце концов они понимают, что им необходимо внедрить понимание предметной области в свои возможности поиска, и в этот момент организации начинают инвестировать в списки синонимов, таксономии, списки известных сущностей и домен-специфичные бизнес-правила. Хотя все это помогает, организации в конечном итоге также обнаруживают, что релевантный поиск во многом зависит от успешной интерпретации пользовательских запросов и понимания намерений пользователя, поэтому они начинают инвестировать в методы классификации запросов, семантического анализа запросов, графы знаний, персонализацию и другие попытки правильно интерпретировать запросы пользователей.

Поскольку эти задачи приводят к улучшениям, этот успех часто приводит к созданию больших команд, вкладывающих значительное время в ручную настройку списков и параметров, и в конечном итоге организации могут понять, что возможно (и более целесообразно) автоматизировать как можно большую часть этого процесса посредством обучения на основе пользовательских сигналов, пользовательского тестирования (A/B-тестирование, моделирование релевантно-

¹ Бустинг в программировании – метод коллективного обучения, при котором модель обучается последовательно, и каждая новая модель пытается исправить предыдущую модель. Несколько слабых моделей объединяются для создания сильной модели, при этом каждая следующая модель обучается на ошибках предыдущих моделей. – *Прим. ред.*

сти в автономном режиме и активное обучение) и построения машинно-обучаемых моделей релевантности. Конечная цель – полностью автоматизировать каждый из этих шагов по ходу развития поисковой разведки и позволить системе быть самообучающейся.

1.3.1. Основа поиска

Первым шагом в создании поисковой платформы почти всегда является запуск традиционного поиска по ключевым словам (часть «понимание контента» на рис. 1.7). Команды часто тратят годы на настройку и улучшение этого шага, и возникла целая дисциплина, называемая *инженерией релевантности*, которая исторически сосредоточила значительные усилия на понимании контента; улучшении контента для поиска; настройке бустинга, параметров запроса и функций запроса и другими способами пытается максимизировать релевантность традиционного опыта поиска. Для глубокого погружения в этот мир инженерии релевантности и настройки традиционной релевантности поиска по ключевым словам мы рекомендуем книгу «Релевантный поиск» Дага Тернбулла и Джона Берримана (Manning, 2016).

По мере того как инженеры по релевантности становятся все более искусными, их работа часто переходит в сферы понимания и рекомендаций пользователей, а также в понимание предметной области и семантического поиска. Рост больших языковых моделей в последние годы облегчил реализацию готового семантического поиска, но выход на следующий уровень оптимизации релевантности и соответствия требует гораздо более сложных подходов, как вы узнаете из этой книги. Наше внимание в этой книге будет сосредоточено на автоматизации процесса обучения и оптимизации релевантности поиска, чтобы он работал как непрерывный цикл обратной связи. По сути, мы хотим автоматизировать большую часть работы инженера по релевантности, полагаясь на алгоритмы, где это возможно, для постоянного изучения оптимальных стратегий соответствия и ранжирования.

Итак, какие характеристики отличают хорошо настроенную поисковую систему от поисковой системы на базе ИИ? Хорошо настроенный поисковый движок является основой, на которой строится поиск на основе ИИ, но поиск на основе ИИ выходит далеко за рамки этого, постоянно обучаясь и совершенствуясь посредством отраженного интеллекта. *Отраженный интеллект* – это идея использования непрерывных циклов обратной связи пользовательского ввода, обновлений контента и взаимодействия пользователя с контентом для постоянного обучения и улучшения качества вашего поискового приложения.

1.3.2. Отраженный интеллект в петлях обратной связи

Циклы обратной связи имеют решающее значение для создания поискового решения на основе ИИ. Представьте, что все ваше образование (от начальной школы до высшей) состояло бы только из чтения учебников: никаких учителей, которые могли бы задавать вопросы, никаких экзаменов для проверки ваших знаний и предоставления обратной свя-

зи, и никаких одноклассников или других людей, с которыми можно было бы взаимодействовать, учиться или сотрудничать. Вы бы, вероятно, наткнулись на бесконечные стены, где не смогли бы полностью понять определенные понятия или даже понять то, что вы читаете, и вы бы неправильно поняли многие идеи и никогда не имели бы возможности осознать это или скорректировать свои предположения.

Поисковые системы часто работают таким же образом. Умные инженеры отправляют данные в поисковую систему и настраивают определенные функции и веса функций, но система просто считывает эти конфигурации и действует на них одинаково каждый раз для повторяющихся пользовательских запросов. Поисковые системы являются идеальным типом системы для интерактивного обучения, однако, когда мы вводим циклы обратной связи.

Рисунок 1.9 показывает типичный поток информации через цикл обратной связи поиска. Сначала пользователь отправляет запрос. Этот запрос выполняет поиск, который возвращает результаты, такие как конкретный ответ, список ответов или список ссылок на страницы, конечному пользователю. После того как ему представлен список, пользователь затем выполняет одно или несколько действий. Эти действия обычно начинаются с кликов по документам, но эти клики в конечном итоге могут привести к добавлению продукта в корзину и его покупке (электронная коммерция), выставлению этому продукту отметки «нравится» или «не нравится» (сайт потребления медиа), лайку или комментированию результата (сайт социальных сетей) или любому количеству других действий, зависящих от контекста.

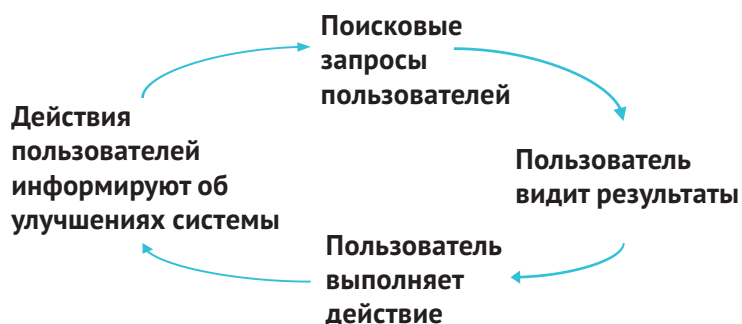


Рис. 1.9. Отраженный интеллект в петлях обратной связи

Эти действия затем можно использовать для создания улучшенной модели ранжирования релевантности для будущих поисков. Ваше поисковое приложение может автоматически корректировать ранжирование будущих результатов поиска, обеспечивая улучшенный поисковый опыт для поиска следующего пользователя.

1.3.3. Бустинг сигналов, совместная фильтрация и обучение ранжированию

Поиски, клики, отметки «нравится», добавление в корзину, покупки, комментарии и другие взаимодействия с вашим поисковым приложением являются критически важными данными, которые вам не-

обходимо фиксировать. Мы совместно называем эти точки данных *сигналами*. Сигналы обеспечивают постоянный поток обратной связи для вашего поискового приложения, записывая каждое значимое взаимодействие с вашими конечными пользователями. Эти цифровые моменты затем могут использоваться алгоритмами машинного обучения для создания моделей для улучшения понимания пользователем, понимания контента и понимания предметной области.

Рисунок 1.10 показывает поток данных для сбора и обработки сигналов в типичном поисковом приложении на базе ИИ. Вы можете видеть сигналы, собираемые для каждого поиска, а также полученные клики и покупки. Уникальные сигналы также могут быть записаны для любого другого вида взаимодействия пользователя (добавление в корзину, фасет-клик, закладка, наведение или даже время пребывания на странице).

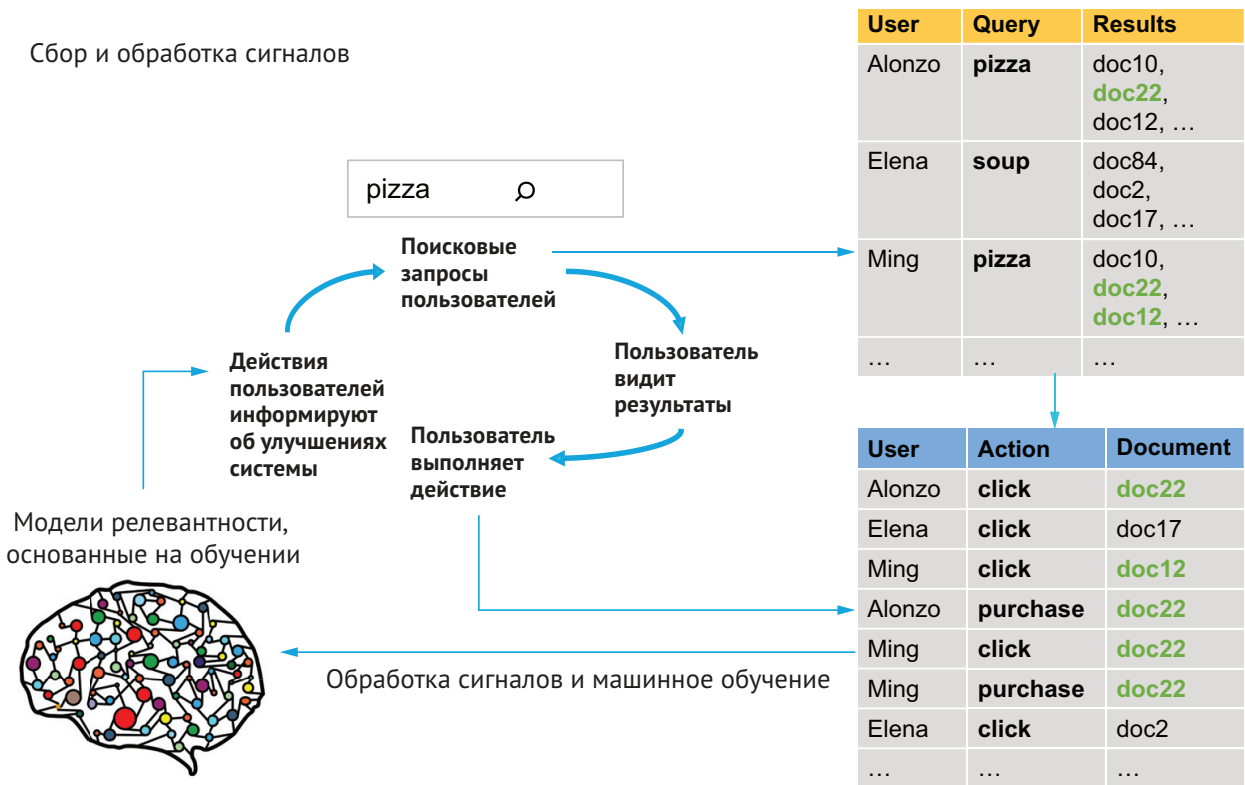


Рис. 1.10. Поток данных сбора и обработки сигналов

Сигналы являются одним из двух источников данных, которые питают интеллектуальный механизм поискового приложения на базе ИИ, а другим является контент. Многие алгоритмы поиска на базе ИИ включают в себя циклы обратной связи сигналов для построения моделей отраженного интеллекта. Некоторые из этих ключевых типов алгоритмов отраженного интеллекта включают:

- популяризированную релевантность – алгоритмы *бустинга сигналов* создают модели, которые используют агрегированные сигналы для повышения рейтинга самых важных документов по вашим самым популярным запросам;

- персонализированную релевантность – алгоритмы *совместной фильтрации* создают модели с использованием матричной факторизации или аналогичных методов, которые используют сигналы для генерации рекомендаций и профилей пользователей для персонализации результатов поиска для каждого пользователя;
- обобщенную релевантность – алгоритмы *обучения ранжированию* обучают классификаторы ранжирования выполнять машинное ранжирование на основе суждений¹ о релевантности, полученных из моделей кликов на основе сигналов пользователя. Этот процесс изучает набор признаков и весов ранжирования, которые можно применять в целом ко всем запросам – даже к тем, которые ранее не встречались.

Эти алгоритмы позволяют вашему поисковому приложению учиться на взаимодействиях с пользователем и автоматически корректировать рейтинги для будущих результатов поиска, обеспечивая улучшенный опыт поиска для следующих поисков пользователей.

1.3.4. Интеллект контента и предметной области

В то время как сигналы обеспечивают постоянный поток данных об использовании и обратной связи для вашего поискового приложения, ваш контент также является богатым источником информации, которая может быть включена в ваши циклы обратной связи. Например, если кто-то ищет определенное ключевое слово, другие ключевые слова и главные категории в возвращенных документах служат ценными точками данных. Эти точки данных могут использоваться для тегирования или категоризации запроса и могут быть показаны другим конечным пользователям (например, как фасеты², что приводит к дальнейшим взаимодействиям, генерирующим сигналы, из которых может учиться поисковый движок.

Контент ваших документов формирует репрезентативную текстовую модель вашего домена. Сущности, доменно-специфическая терминология и предложения, содержащиеся в ваших документах, служат богатым семантическим графом. Этот граф можно использовать для управления мощным понятийным и семантическим поиском, который лучше понимает ваш домен. Мы более подробно рассмотрим понимание вашего контента в главе 2 и возможности семантического поиска с использованием этого богатого семантического графа знаний (SKG) в главе 5.

¹ Суждения, англ. *judgments*, в программировании связаны с понятием логического программирования. Это парадигма программирования, основанная на математической логике, в которой программы задаются в форме логических утверждений и правил вывода. – *Прим. ред.*

² Фасетный поиск (также фасетная навигация или фасетный просмотр) – метод доступа к информации с использованием одновременно нескольких фильтров – фасетов. Организуется в соответствии с фасетной классификацией. Для использования фасетного поиска каждая информационная единица классифицируется по нескольким явным характеристикам (фасетам). Фасетный поиск противопоставляется таксономическому порядку фильтрации и упорядочивания. – *Прим. ред.*

В последние годы LLM произвели революцию в том, как поисковые системы могут интерпретировать запросы и ответы. LLM – это глубокие нейронные сети, обученные на огромных объемах текстовых данных. Они могут распознавать, переводить, обобщать, предсказывать и генерировать новые данные на основе входящих подсказок и любого дополнительного предоставленного контекста. Часто LLM обучаются на тексте, получают подсказку в виде текста и возвращают ответ в виде текста, хотя подобные мультимодальные модели также могут обучаться на изображениях, аудио, других данных или на всем вышеперечисленном. LLM часто содержат миллиарды параметров в нейронной сети, и это число, вероятно, продолжит расти в будущем, пока производительность модели продолжает улучшаться с большим количеством параметров.

Сегодняшние самые успешные LLM основаны на архитектуре трансформеров, представленной исследователями Google в 2017 году, которая применяет концепцию «внимания» к изучению языка («Внимание – это все, что вам нужно», Ашиш Васвани с соавт.). Огромные объемы текстовых данных подаются в нейронную сеть, а представление слов и их отношений в каждом контексте моделируется с использованием неконтролируемого обучения. После того как модель построена, она способна интерпретировать входящую строку текста, *подсказку*, как контекст и кодировать контекст в эмбединги, которые являются числовыми векторными представлениями смысла подсказки. Помимо возможности кодировать подсказки в эмбединги, трансформеры также содержат слой декодировщика, который может преобразовывать эмбединги обратно в текст. Трансформеры могут использоваться для решения многих видов задач, от поиска сходства в эмбедингах (поиск текста, поиск изображений и т. д.) до ответов на вопросы, классификации, обобщения контента и даже создания нового контента (письма, кода, стихотворений, изображений и т. д.).

Трансформеры зависят от контекста. LLM, настроенный на генерацию ответов на вопросы, может ответить на вопрос «What is the difference between a capital and capitol?» (В чем разница между столицей и капитолием?) ответом «Capital is a city or town that serves as the seat of government for a state or country. A capitol is a building in which a state legislature meets» (Столица – это город или поселок, который служит резиденцией правительства штата или страны. Капитолий – это здание, в котором заседает законодательный орган штата). Однако тот же LLM может ответить на вопрос «What is the difference between a capital and lowercase word?» (В чем разница между заглавными и строчными буквами?) следующим ответом, основанным на контексте: «The difference between a capital and lowercase word is that a capital letter is used at the beginning of a sentence or proper nouns, while a lowercase letter is used for all other letters in a word» (Разница между капителью и строчными словами заключается в том, что заглавная буква используется в начале предложения или имен собственных, в то время как строчная буква используется для всех остальных букв в слове). Многие LLM

имеют открытый исходный код, но для оптимального качества вывода LLM выигрывают от тонкой настройки под задачу с помощью предметно-специфического контента и подсказок. Тонкая настройка – это процесс взятия предварительно обученной модели, которая уже имеет сильное общее понимание языка и общих понятий, и «обучения» ее новому контенту и задачам. Исходные предварительно обученные модели часто называют *базовыми моделями*¹, поскольку они формируют основу, на которой будет применяться предметно-специфическая тонкая настройка. Процесс тонкой настройки обычно занимает малую часть времени, необходимого для обучения исходного LLM. Некоторые LLM были обучены на таком большом количестве данных и таком широком разнообразии данных (например, комплексном веб-сканировании интернета), что они могут работать довольно хорошо без переобучения, но переобучение под задачу почти всегда повышает производительность.

1.3.5. Генеративный ИИ и RAG

Генеративный ИИ ускоряется быстрыми темпами, и поисковые системы извлекают из этого выгоду и служат ключевым компонентом систем генеративного ИИ. LLM (и другие базовые модели) служат в качестве рассуждающих машин, имея достаточно знаний о мире, чтобы интерпретировать язык и в целом рассуждать о большинстве вопросов, но без возможности надежно вспоминать фактическую информацию без риска галлюцинаций (создания ложной информации).

В результате поисковые системы используются в конвейерах генерации, дополненной результатами поиска (RAG) в качестве источника знаний для LLM, позволяя извлекать соответствующий контекст и передавать его LLM, чтобы гарантировать, что у него есть актуальные и точные данные для ответа. Вся эта книга фактически посвящена использованию ИИ для оптимизации «поисковой» части RAG, а «генеративную» часть мы рассмотрим в главе 15.

В то время как RAG делает поисковые движки критически важным компонентом генеративных систем ИИ, LLM также служат критически важными компонентами поисковых систем. LLM могут использоваться для интерпретации запросов, генерации эмбедингов для векторного поиска, генерации резюме результатов поиска и даже генерации ответов на вопросы непосредственно из результатов поиска.

Переход от традиционного поиска информации к этим новым возможностям *генеративного поиска* показан на рис. 1.11. В течение десятилетий традиционный поиск возвращал список результатов поиска («десять синих ссылок»), показывая документы с самым высоким рейтингом, наиболее релевантные запросу. Для запросов по сущностям

¹ Базовая модель, англ. *foundation model*, представляет собой модель машинного обучения или глубокого обучения, которая обучается на обширных наборах данных, поэтому ее можно применять в широком диапазоне вариантов использования. Приложения с генеративным ИИ, такие как большие языковые модели, часто являются примерами базовых моделей. – Прим. ред.

и известным темам поисковые системы часто показывают предварительно рассчитанные информационные поля с краткой информацией или показывают заранее определенные ответы на известные вопросы. Поисковые системы часто также извлекают слова, предложения или фрагменты абзацев из результатов поиска, чтобы ответить на вопросы, вместо того чтобы заставлять пользователей открывать и читать результаты поиска, чтобы найти ответ. Этот процесс известен как *экстрактивный метод ответа на вопрос*¹, и это более целенаправленная форма поиска, поскольку он дополнительно ищет и ранжирует ответы, найденные в документах.



Рис. 1.11. Переход от традиционного поиска информации к генеративному поиску

Однако существует тонкая грань между извлечением ответов из результатов поиска и синтезом нового контента для возврата в результатах, и именно здесь мы переходим в сферу генеративного поиска. *Суммаризация результатов* – это процесс переписывания результатов поиска в более краткий и читабельный формат, часто объединяющий информацию из нескольких источников и даже предоставляющий ссылки на источники в обобщенном ответе. *Генерация ответов на абстрактные вопросы* – это процесс генерации ответов на вопросы путем синтеза информации из одного или нескольких ранжированных результатов поиска в ответ на вопрос пользователя. Разница между генерацией ответов на извлекаемые вопросы и генерацией ответов на абстрактные вопросы заключается в том, что первая находит релевантный контент в документах для возврата в качестве ответов («извлекает его»), тогда как вторая пишет синтезированный ответ, интерпретируя результаты и генерируя ответ, который может отличаться от того, что написано в любом из документов. *Генерация нового контента* также возможна в рамках генеративного поиска, например ответ на запросы с помощью креативной новой прозы, кода, стихотворений, изображений или другого контента на основе ключевых слов или подсказок, отправляемых пользователями. Подводя итог, можно сказать, что генеративный ИИ и поиск на основе ИИ тесно переплетены. Генеративный ИИ является критически важным компонентом «поиска на основе ИИ»

¹ Экстрактивный метод ответа на вопрос, англ. *extractive question answering*, в программировании – это задача, в которой модель извлекает ответ на вопрос из заданного контекста. – Прим. ред.

(обеспечивает генерацию ответов и обобщение результатов), а поиск на основе ИИ является критическим компонентом «поискового ИИ» (RAG). Оба активно используют LLM и другие базовые модели, и оба являются важнейшими компонентами интеллектуальных и точных систем искусственного интеллекта.

1.3.6. Контролируемый искусственный интеллект в сравнении с искусственным интеллектом «черного ящика»

Как и LLM, многие современные методы ИИ в значительной степени полагаются на глубокое обучение на основе искусственных нейронных сетей. К сожалению, человеку часто сложно понять конкретные факторы, которые входят в любой конкретный прогноз или вывод из модели глубокого обучения из-за внутренней сложности изученной модели.

Иногда это приводит к системе ИИ «черного ящика»¹, где результаты могут быть правильными или впечатляющими, но их нелегко отладить или исправить, когда модель выносит неверное суждение. Целая область *контролируемого ИИ* (иногда называемого *объяснимым ИИ*, *курируемым ИИ* или *прозрачным ИИ*) возникла из-за необходимости понимать, курировать и доверять этим моделям.

В этой книге мы рассмотрим подходы глубокого обучения к поиску, такие как поиск по плотным векторам по эмбедингам, генерацию ответов на вопросы, генерацию синтетических обучающих данных и обобщение результатов с использованием LLM. Однако мы в основном сосредоточим наши усилия на создании интеллекта, который может быть выражен в человеческих терминах, а затем исправлен и дополнен человеческим интеллектом. Вы можете думать об этом как о «человеческом контроле с помощью ИИ» или как о «человеческом ИИ», но в любом случае основная философия этой книги заключается в использовании ИИ для автоматизации процесса интеллектуального поиска, при этом человек остается в курсе событий с возможностью взять под контроль и дополнить или переопределить систему.

В качестве учебного упражнения этот подход также приводит к более глубокому, интуитивному пониманию того, как работает ранжирование и релевантность поиска и как можно интегрировать множество различных подходов, основанных на ИИ, не теряя контроля над системой.

1.3.7. Архитектура поисковой системы на основе ИИ

Архитектура поисковой системы (движка) на основе ИИ часто требует сборки множества строительных блоков для формирования интеллектуальной сквозной системы. Вы начинаете с базовой поисковой

¹ «ИИ черного ящика» – это системы искусственного интеллекта, внутренняя работа которых скрыта от пользователя. Такие модели предназначены для обработки больших объемов данных и автономного изучения сложных закономерностей. Они производят вывод на основе заданного ввода, но внутренняя работа модели неизвестна даже разработчикам. – Прим. ред.

системы, например Apache Solr, OpenSearch или одной из других поисковых систем или векторных баз данных, указанных в приложении В. Затем вы загружаете в систему свой доступный для поиска контент, выполняя различные преобразования, чтобы сделать его более полезным. Эти преобразования времени индексирования могут включать такие изменения:

- интерпретацию смысла ваших документов в эмбединги с использованием LLM;
- классификацию документа, добавление классификации в качестве поля;
- нормирование значений полей;
- извлечение сущностей из текста, добавление сущностей в отдельные поля;
- кластеризацию контента, добавление кластеров в качестве поля;
- обнаружение и аннотирование фраз;
- извлечение дополнительных данных из графа знаний, внешнего API или другого источника данных;
- выполнение обнаружения части речи (POS) и других шагов обработки естественного языка;
- извлечение фактов (например, триплетов RDF);
- применение других моделей машинного обучения или правил ETL для обогащения документа.

После того как данные находятся в движении, ваша цель – сделать их доступными для поиска. Для этого требуются конвейеры запросов, которые могут интерпретировать входящие запросы; определять понятия, фразы и сущности; исправлять опечатки; расширять запрос, включая в него связанные термины, синонимы, понятия или эмбединговые представления; а затем переписывать запрос так, чтобы ваш основной движок мог найти наиболее релевантные результаты. Отдельные документы поиска могут затем быть возвращены конечному пользователю, сводки результатов могут быть сгенерированы из языковых моделей, или ответы могут быть явным образом извлечены из результатов.

Однако большая часть этого интеллекта запроса требует надежного понимания вашего домена. Это требует запуска пакетных заданий на вашем контенте и пользовательских сигналах для изучения закономерностей и получения интеллекта, специфичного для домена. Каковы наиболее распространенные ошибки написания от ваших пользователей, и что они выбирают в качестве правильного написания среди нескольких вариантов? Когда пользователь ищет определенные запросы, какие документы следует повысить в рейтинге как самые популярные? Для неизвестных запросов каков идеальный рейтинг среди всех атрибутов или признаков, доступных для сопоставления?

Нам нужен доступ к большинству этих ответов во время запроса (либо предварительно вычисляемый, либо быстро вычисляемый), поскольку мы ожидаем, что запросы будут возвращаться в течение

миллисекунд или секунд. Для этого требуется фреймворк¹ обработки заданий (в этой книге мы используем Apache Spark) и механизм планирования рабочих процессов для поддержания последовательности выполнения заданий.

Вам также понадобится механизм для сбора постоянного потока входящих сигналов пользователей – захват их в приложении пользовательского интерфейса (frontend) и последующее сохранение в вашей поисковой системе или другом хранилище данных (backend).

Затем сигналы будут использоваться для генерации всех видов моделей – от моделей бустинга сигналов, которые повышают самые популярные предметы для самых популярных запросов, до моделей обучения ранжированию, которые применяют обобщаемые функции ранжирования ко всем запросам, до моделей персонализации, которые выводят пользовательские рекомендации и предпочтения персонализации для каждого пользователя или сегмента пользователей.

Поиск на основе ИИ – это гораздо больше, чем просто использование новейшей LLM для интерпретации запросов. Речь идет о разработке сквозной системы для непрерывного обучения. В конечном итоге вы получите систему, которая получает постоянные потоки изменений документов и сигналов пользователей, постоянно обрабатывает эти потоки для улучшения моделей, а затем постоянно корректирует будущие результаты поиска и измеряет влияние изменений, чтобы предоставлять более интеллектуальные результаты. Это ключ к поиску на основе ИИ: реализация процесса непрерывного обучения и совершенствования на основе реальных взаимодействий пользователей, обновление шаблонов контента и развитие моделей для оптимального понимания текущих намерений пользователя и предоставления постоянно улучшающегося опыта поиска.

Резюме

- Ожидания в отношении сложности поиска развиваются с ростом LLM, при этом конечные пользователи ожидают, что поиск теперь будет предметно-ориентированным, контекстным и персонализированным, разговорным, мультимодальным, интеллектуальным и вспомогательным.
- Поиск и рекомендации – это два крайних конца непрерывного спектра персонализации в поиске информации, и важно учитывать возможности между ними для оптимизации релевантности.
- Правильная интерпретация намерений пользователя требует одновременного понимания вашего контента, вашего пользова-

¹ Фреймворк в программировании (англ. *framework* – каркас, структура) – это набор инструментов, компонентов и методов, которые облегчают разработку программного обеспечения. Простыми словами, это готовый шаблон для написания программы, который помогает решать следующие задачи: увеличение скорости разработки продукта (использование готовых модулей позволяет сократить время работы и ускорить выпуск продукта), снижение количества ошибок в коде. – *Прим. ред.*

теля и его предпочтений, а также области знаний, в которой работает ваша платформа.

- Оптимальная релевантность поиска лежит на пересечении персонализированного поиска (традиционный поиск по ключевым словам плюс совместные рекомендации), семантического поиска (традиционный поиск по ключевым словам плюс графы знаний) и рекомендаций с учетом домена (совместные рекомендации плюс графы знаний).
- Поиск на основе ИИ работает и обучается на двух ключевых типах данных: контенте и сигналах пользователя.
- Поиск и генеративный ИИ идут рука об руку. Возможности генеративного поиска, такие как RAG, являются критически важным компонентом современных систем генеративного ИИ (для предотвращения галлюцинаций); а возможности генеративного ИИ, такие как суммирование результатов, являются критически важными компонентами современных поисковых систем (для возврата лучших ответов).
- Отраженный интеллект – использование циклов обратной связи для постоянного сбора сигналов, настройки результатов и измерения улучшений – это двигатель, который позволяет поиску на основе ИИ учиться и постоянно совершенствоваться.

Работа с естественным языком

https://t.me/it_books/2

В этой главе рассматриваются:

- скрытые структуры в неструктурированных данных;
- философия языка, ориентированная на поиск;
- изучение распределительной семантики и векторных эмбедингов;
- моделирование домен-специфичных знаний;
- проблемы с естественным языком и запросами;
- применение методов понимания естественного языка как к контенту, так и к сигналам.

В первой главе мы представили общий обзор того, что значит создать поисковую систему на основе ИИ. В оставшейся части книги мы рассмотрим и продемонстрируем многочисленные способы, которыми ваше поисковое приложение может непрерывно учиться на вашем контенте и поведенческих сигналах ваших пользователей, чтобы лучше понимать ваш контент, ваших пользователей и ваш домен и в конечном итоге предоставлять пользователям необходимые им ответы. Мы получим гораздо больше практических знаний в главе 3, запустив поисковый сервер по вашему выбору и уровень обработки данных (Apache Spark) и начав с первого из наших блокнотов Jupyter, который мы будем использовать на протяжении всей книги, чтобы пройти через множество пошаговых примеров.

Однако, прежде чем мы погрузимся в эти практические примеры и конкретные реализации, в этой главе важно сначала установить общую ментальную модель для высокоуровневых задач, которые мы пытаемся решить. В частности, когда дело доходит до интеллектуального поиска, нам приходится иметь дело со многими сложностями и нюансами в естественном языке – как в контенте, который мы ищем, так и в поисковых запросах наших пользователей. Нам приходится иметь дело с ключевыми словами, сущностями, понятиями, опечатками, синонимами, аббревиатурами, инициализмами, неоднозначными терминами, явными и подразумеваемыми отношениями между понятиями, иерархическими отношениями, обычно встречающимися в таксономиях, высокоуровневыми отношениями, обычно встречающимися в онтологиях, и конкретными примерами отношений сущностей, обычно встречающимися в комплексных графах знаний.

Хотя может возникнуть соблазн сразу же погрузиться в некоторые конкретные задачи, например как автоматически находить орфографические ошибки в контенте или как распознавать синонимы в сеансах поиска пользователей, будет более разумно сначала установить концептуальную основу, которая объясняет, с какими *типами* проблем нам приходится иметь дело при поиске и в понимании естественного языка (NLU). Установление этой философской основы позволит нам создавать лучшие комплексные решения в нашей поисковой системе на базе ИИ, где все части будут работать вместе сплоченным и интегрированным образом. Таким образом, в этой главе будут предоставлены философские основы того, как мы решаем проблемы понимания естественного языка на протяжении всей этой книги и как мы применяем эти решения, чтобы сделать наши поисковые приложения более интеллектуальными.

Начнем с обсуждения некоторых распространенных заблуждений о природе свободного текста и других неструктурированных источников данных.

2.1. Миф о неструктурированных данных

Термин «неструктурированные данные» использовался в течение многих лет для описания текстовых данных, покуда они не отформатированы таким образом, чтобы их можно было легко интерпретировать и запрашивать. Однако широко распространенное представление о том, что текст или любые другие данные, которые не соответствуют заранее определенной схеме («структуре»), на самом деле «не структурированы», является мифом, на пересмотр которого мы потратим время в этом разделе.

Если вы посмотрите, что такое *неструктурированные данные* в «Википедии», увидите, что они определяются как «информация, которая не соответствует заранее определенной модели данных и, как правило, представлена в форме текста с датами, числами, фактами, расположенными в нем в произвольной форме». Далее в статье го-

ворится, что «такие данные трудно анализировать, особенно при помощи традиционных программ, предназначенных для работы со структурированными данными (аннотированными или хранящимися в базах)».

Однако фраза «неструктурированные данные» – неудачный термин для описания текстового контента. В действительности термины и фразы, присутствующие в тексте, кодируют огромное количество смысла, а лингвистические правила, применяемые к тексту для придания ему смысла, служат его собственной структурой. Назвать текст неструктурированным – это немного похоже на то, чтобы назвать песню, играющую по радио, «произвольными звуковыми волнами». Несмотря на то что каждая песня имеет уникальные характеристики, большинство из них демонстрируют общие атрибуты (темп, мелодию, гармонию, текст и т. д.). Хотя эти атрибуты могут отличаться или отсутствовать в той или иной песне, они тем не менее соответствуют общим ожиданиям, которые позволяют передавать и извлекать смысл из каждой песни. Текстовая информация обычно следует схожим правилам – структуре предложения, грамматике, пунктуации, взаимодействию между частями речи и т. д. На рис. 2.1 показан пример текста, который мы рассмотрим немного подробнее в следующих разделах, когда будем исследовать эту структуру.

Трей Грейнджер работает в Searchkernel.
Он выступал на конференции Haystack 2023.
#haystackconf (Haystack) проходила в Шарлоттсвилле,
штат Вирджиния, 25–26 апреля 2023 года. Трей получил
степень магистра в Georgia Tech.

Рис. 2.1. Неструктурированные данные. Этот текст представляет собой типичные неструктурированные данные, которые вы можете найти в поисковой системе

Хотя текст является наиболее распространенным типом неструктурированных данных, существует несколько других видов неструктурированных данных со схожими характеристиками, как мы увидим в следующем разделе.

2.1.1. Типы неструктурированных данных

Свободный текстовый контент считается основным типом неструктурированных данных, но поисковые системы обычно индексируют многие другие виды данных, которые также не вписываются в структурированную базу данных. Распространенными примерами являются изображения, аудио, видео и журналы событий. Рисунок 2.2 расширяет наш текстовый пример из рис. 2.1 и включает несколько других типов неструктурированных данных, таких как аудио, изображения и видео.

Аудио наиболее похоже на текстовый контент, поскольку часто это просто еще один способ кодирования слов и предложений. Конечно, аудио может включать в себя гораздо больше, чем просто произнесенные слова, – оно может включать в себя музыку и неязыковые звуки, и оно может более эффективно кодировать такие нюансы, как эмоции, тон голоса и одновременно перекрывающееся общение.

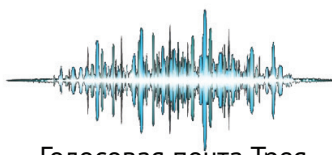
Неструктурированные данные



Трей Грейнджер работает
в Searchkernel.
Он выступил на конференции
Haystack 2023.

#haystackconf (Haystack) проходила
в Шарлоттсвилле, штат Вирджиния,
25–26 апреля 2023 года.

Трей получил степень магистра
в Georgia Tech.
Голосовая почта Трея



Голосовая почта Трея



Рис. 2.2. Несколько типов неструктурированных данных. Помимо текста с последнего рисунка, мы теперь видим изображения, аудио и видео, которые являются другими формами неструктурированных данных

Изображения – это еще один вид неструктурированных данных. Так же как слова образуют предложения и абзацы для выражения идей, они содержат сетки цветных пикселей, которые, взятые вместе, образуют изображение.

Видео, таким образом, служит еще одним видом неструктурированных данных, поскольку оно представляет собой временную последовательность изображений, а также необязательного звука, который совпадает с этим видеорядом.

Когда неструктурированные данные обнаруживаются смешанными со структурированными данными, мы обычно называем это *полуструктурированными* данными. Данные журнала являются прекрасным примером таких полуструктурированных данных. Часто сообщения журнала содержат структурированную дату события, структурированные типы событий (например, предупреждение или ошибка, поиск или клик), а затем какое-то неструктурированное сообщение или описание в свободном тексте.

Технически говоря, практически любой тип файла можно считать неструктурированными данными, но мы в первую очередь будем иметь дело с вышеупомянутыми типами. Поисковым системам часто поручают обработку каждого из этих видов неструктурированных данных, поэтому мы обсудим стратегии их обработки на протяжении всей книги.

2.1.2. Типы данных в традиционных структурированных базах данных

Чтобы лучше работать с нашими неструктурированными данными, может быть полезно сначала сравнить их со структурированными данными в базе данных SQL. Это позволит нам позже провести параллели между тем, как мы можем запрашивать неструктурированные представления данных по сравнению со структурированными.

Запись (строка) в базе данных SQL сегментируется на столбцы, каждый из которых может содержать определенный тип данных. Некоторые из этих типов данных представляют собой дискретные значения – значения, которые берутся из перечисленного списка, такие как идентификаторы, имена или текстовые атрибуты. Другие столбцы могут содержать непрерывные значения, такие как диапазоны даты/времени, числа и другие типы столбцов, которые представляют собой диапазоны без конечного числа возможных значений.

В общем, когда нужно связать разные строки вместе или связать их со строками в других таблицах базы данных, «объединения» (joins) будут выполняться над дискретными значениями. *Объединения* используют общее значение (часто поле идентификатора) для связывания двух или более записей вместе для формирования составной записи.

Например, если у кого-то есть две таблицы данных, одна из которых представляет сотрудников, а другая – компании, то, скорее всего, в таблице `companies` будет столбец `id`, а в таблице `employees` – соответствующий столбец `company_id`. Поле `company_id` в таблице `employees` называется *внешним ключом* (foreign key), т. е. значением в одной таблице, которое ссылается на сущность в другой таблице, связывая записи на основе общего идентификатора. Рисунок 2.3 демонстрирует это, показывая примеры дискретных значений, непрерывных значений и объединения таблиц с использованием внешнего ключа.

Структурированные данные

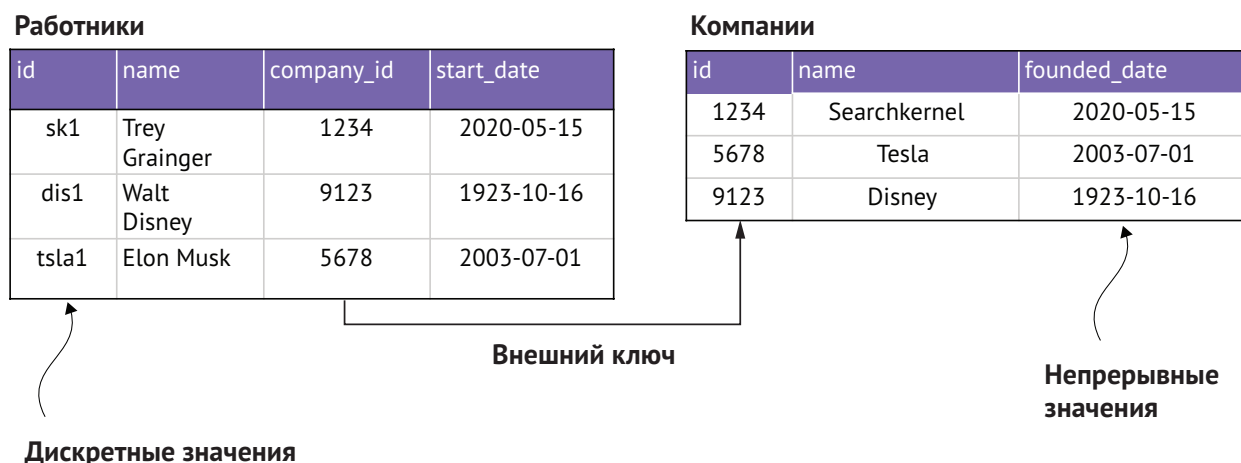


Рис. 2.3. Структурированные данные в типичной базе данных. Дискретные значения представляют идентификаторы и перечисляемые значения, непрерывные значения представляют данные, которые попадают в диапазоны, а внешние ключи существуют, когда одно и то же значение существует в двух таблицах и, таким образом, может использоваться как общий атрибут, который создает связь между соответствующими строками из каждой таблицы

Это представление объединения различных записей на основе известных отношений (ключей и внешних ключей) является мощным способом работы с реляционными данными в явно смоделированных таблицах, но, как мы увидим в следующем разделе, очень похожие методы можно применять и к неструктурированным данным свободной формы.

системе, каждый из которых содержит значение «Haystack». В обоих случаях мы можем считать эти документы связанными внешним ключом.

Нечеткие внешние ключи в неструктурированных данных

Однако с неструктурированными данными у нас гораздо больше возможностей, чем с традиционным моделированием структурированных данных. Например, на рис. 2.5 обратите внимание, что теперь два документа связаны и оба ссылаются на ведущего автора этой книги – один использует мое полное имя «Тре́й Грейнджер», а другой просто использует мое имя «Тре́й».

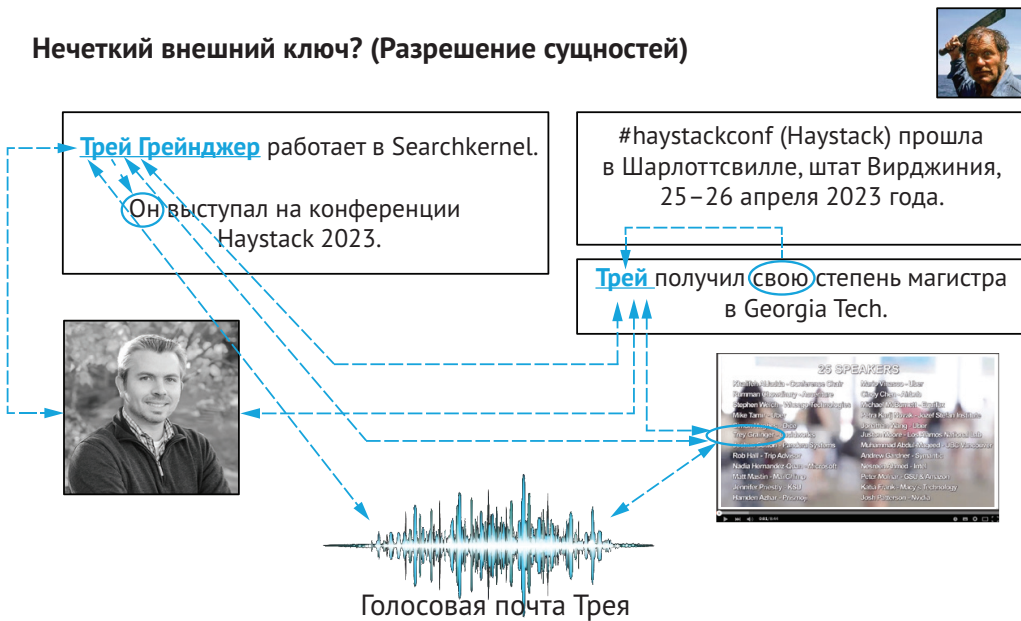


Рис. 2.5. Нечеткие внешние ключи. В этом примере одна и та же сущность упоминается с использованием разных терминов, а соединение происходит на основе нескольких фраз, разрешающихся в одну и ту же сущность

Это пример *разрешения сущности* (entity resolution), где есть два разных представления сущности, но они все равно могут быть разрешены к одному и тому же значению и, следовательно, могут использоваться для объединения информации между двумя документами. Вы можете думать об этом как о «нечетком внешнем ключе», поскольку это все еще внешний ключ, но не в строгом смысле сопоставления токенов¹, так как для разрешения требуются дополнительные методы обработки естественного языка и разрешения сущностей.

Открыв эту дверь расширенной обработки текста для разрешения сущностей, мы можем узнать еще больше из нашей неструктурирован-

¹ Токен в программировании – это минимальная единица кода, которая имеет смысл в контексте языка программирования. Представьте себе токен как слово в предложении: каждое слово имеет значение и играет определенную роль. В программировании токены могут быть ключевыми словами, идентификаторами, операторами, литералами и другими элементами, которые компилятор или интерпретатор языка понимает и обрабатывает. Токены являются основными строительными блоками любого программного кода. – Прим. ред.

ной информации. Например, не только имена «Трей» и «Трей Грейнджер» в этих документах ссылаются на одну и ту же сущность, но также слова «он» и «его».

Вы также заметите, что и мое изображение (в нижнем левом углу на случай, если вы не знаете, как я выгляжу), и видео, содержащее ссылку на мое имя, идентифицированы как связанные и присоединены к текстовым ссылкам. Мы полагаемся на скрытую структуру, присутствующую во всех этих неструктурированных данных, чтобы понять значение, связать документы вместе и узнать еще больше о каждой из упомянутых сущностей в этих документах.

Работа с неоднозначными терминами

Пока все хорошо, но в реальном контенте не всегда уместно предполагать, что один и тот же термин в нескольких местах несет одно и то же значение или даже что наше разрешение сущностей всегда разрешает сущности правильно. Эта проблема одинакового написания слов и фраз, имеющих несколько потенциальных значений, называется *полисемией* (polysemy), и работа с этими неоднозначными терминами может стать огромной проблемой в поисковых приложениях.

Вы могли заметить странное изображение в правом верхнем углу предыдущих рисунков, которое показалось немного неуместным. На этом изображении показан довольно устрашающий человек, держащий мачете. Судя по всему, если вы зайдете в Google и введете запрос Trey Grainger то увидите это изображение. Если вы углубитесь, то увидите на рис. 2.6, что есть пользователь x.com (ранее Twitter) с таким же именем «Trey Grainger», и это изображение – его фотография профиля.

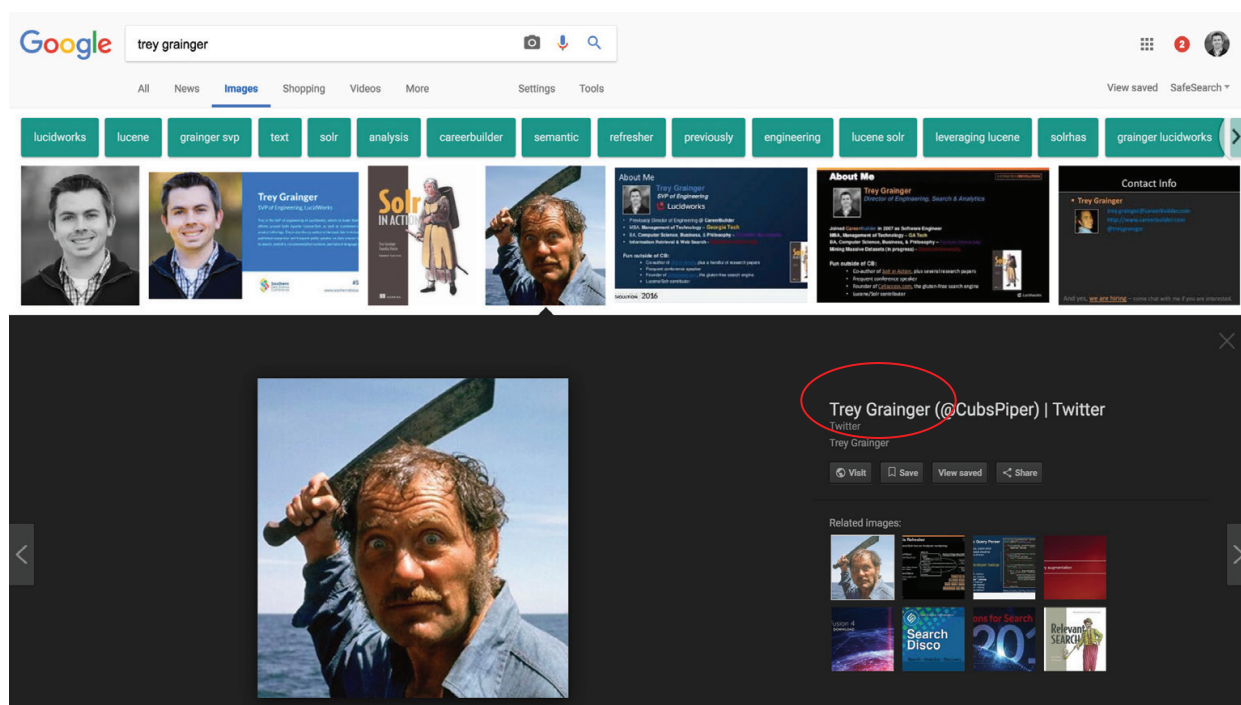


Рис. 2.6. Полисемия. На этом изображении показан поиск Google по запросу Trey Grainger. Возвращаются фотографии нескольких разных людей, потому что имена этих людей имеют одинаковое написание, что делает фразу «Trey Grainger» неоднозначной

На фотографии, по-видимому, Роберт Шоу (который играет Квинта в фильме 1975 года «Челюсти»), но это определенно не то, что вы хотите, чтобы люди первым делом натыкались на вас, когда ищут вас в интернете!

Здесь можно извлечь два важных урока. Во-первых, возможно, никогда не гуглите себя (вы можете ужаснуться тому, что найдете!). Во-вторых, и это более серьезно, полисемия – это серьезная проблема в поиске и понимании естественного языка. Небезопасно предполагать, что термин имеет единственное значение или даже единое значение в разных контекстах, поэтому наша поисковая система на основе ИИ должна использовать контекст, чтобы не путать эти различные значения.

Неструктурированные данные как гигантский граф соотношений

В предыдущих разделах мы увидели, что неструктурированные данные не только содержат богатую информацию (сущности и их отношения), но и что можно связывать различные документы, объединяя их по общим сущностям, подобно тому, как работают внешние ключи в традиционных базах данных. Однако типичные неструктурированные данные содержат так много таких отношений, что вместо того, чтобы думать в терминах строк и столбцов, может быть полезнее думать о наборе данных как о гигантском графе отношений, как мы рассмотрим в этом разделе.

На этом этапе должно быть ясно, что в неструктурированных данных скрыто гораздо больше структуры, чем большинство людей осознает. Неструктурированная информация на самом деле больше похожа на гиперструктурированную информацию – это граф, который содержит гораздо больше структуры, чем типичные структурированные данные.

Рисунок 2.7 демонстрирует этот гигантский граф отношений, который присутствует даже в небольшой горстке документов в нашем примере. Вы можете видеть имена, даты, события, локации, людей, компании и другие сущности, и вы можете вывести связи между ними, используя соединения между сущностями в документах. Вы также заметите, что изображения были правильно устранены, так что парень с мачете теперь отключен от графа. Если все это можно узнать всего из нескольких документов, представьте, что можно узнать из тысяч, миллионов или миллиардов документов, которые есть в вашей поисковой системе.

Часть ценности поисковой платформы на базе ИИ заключается в возможности извлекать такие идеи из ваших данных. Вопрос в том, как использовать этот огромный граф семантических знаний для управления этим интеллектом?

Один из самых эффективных способов использования графа из текстовых данных – это большая языковая модель (LLM), такая как модель трансформера (transformer), которая была представлена в разделе 1.3.4. Эти модели используют глубокое обучение для изучения миллиардов параметров в огромных наборах данных, таких как сканирование (crawls) большей части интернета, для создания детального понимания языка. Это понимание включает как значения слов в разных

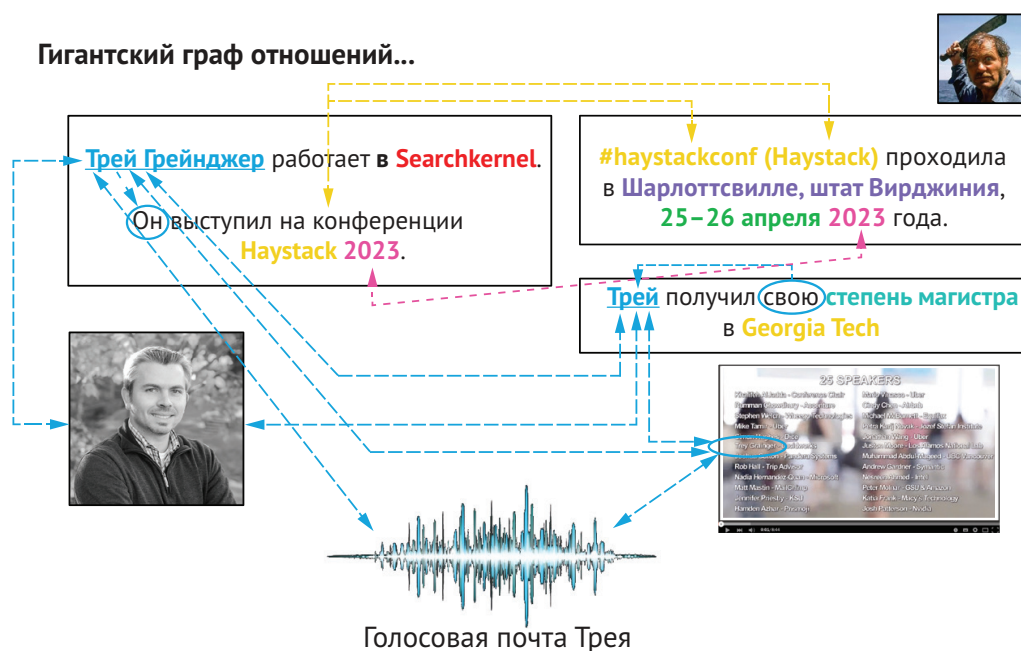


Рис. 2.7. Гигантский граф отношений. Богатый граф взаимосвязей возникает даже из небольшой коллекции связанных документов

контекстах, так и лингвистические и понятийные связи между словами. Эти модели внутренне представляют гигантский граф отношений (graph of relationships), обнаруженных во всех данных, на которых они обучаются, которые обычно более общие, чем ваш набор данных, поэтому модели должны быть точно настроены для изучения любых доменно-специфических отношений в ваших данных. Эта необходимость в тонкой настройке может создать некоторые проблемы, так как LLM являются своего рода «черным ящиком», поскольку в противном случае они не представляли бы ваш набор данных оптимальным образом, и возвращаемая ими информация могла бы быть ошибочной.

К счастью, внутренняя структура инвертированного индекса в вашей поисковой системе позволяет очень легко перемещаться по большому графу отношений в ваших данных без необходимости какого-либо дополнительного явного моделирования данных. *Инвертированный индекс* является основной структурой данных, используемой для лексического поиска, он сопоставляет каждое ключевое слово или термин в полях ваших документов со списками (называемыми *списками постов*) всех документов, содержащих эти ключевые слова. Инвертированный индекс позволяет очень быстро выполнять поиск набора документов, содержащих любой заданный термин (или последовательность терминов, если рассматривать позиционное сопоставление и булеву логику, реализованную посредством операций над множествами). С помощью этих поисков можно перемещаться между различными последовательностями терминов, используя их общие документы для вычисления взвешенного ребра в графе. Мы подробно рассмотрим, как использовать этот семантический граф знаний, скрытый в ваших данных, в главе 5.

2.2. Структура естественного языка

В последнем разделе мы обсудили, как текст и неструктурированные данные содержат гигантский граф отношений, который можно построить, прослеживая общие термины между различными записями. Если вы уже некоторое время занимаетесь разработкой поисковых систем, вы привыкли думать о контенте как о документах, полях и терминах в этих полях. Однако при интерпретации семантического значения вашего контента следует учитывать еще несколько уровней.

Рисунок 2.8 демонстрирует эти дополнительные уровни семантического значения. На самом базовом уровне у вас есть *символы* (characters), которые представляют собой отдельные буквы, цифры или символы, такие как буква «e» на рисунке. Затем один или несколько символов объединяются для формирования *последовательностей символов* (character sequences), таких как «e», «en», «eng», ... «engineer» и «engineers». Некоторые последовательности символов образуют термины, которые представляют собой законченные слова или токены, несущие идентифицируемое значение, например «инженер», «инженеры», «инженерия» или «программное обеспечение». Затем один или несколько терминов можно объединить в *последовательность терминов* – обычно называемую *фразами* (phrases), когда все термины последовательны. К ним относятся такие вещи, как «инженер-программист», «инженеры-программисты» и «старший инженер-программист». Для простоты в этой книге мы также считаем отдельные термины «последовательностями терминов», поэтому всякий раз, когда мы ссылаемся на «фразы», это включает в себя отдельные термины.

Последовательности терминов и фразы

Вы можете задаться вопросом, в чем разница между «последовательностью терминов» и «фразой». Проще говоря, фраза – это последовательность терминов, в которой все термины появляются последовательно. Например, “chief executive officer” – это и фраза, и последовательность терминов, тогда как “chief officer”~2 (что означает «должностное лицо» в пределах двух позиций или расстояний редактирования от «главный») – это только последовательность терминов, поскольку она определяет последовательность терминов, которая не обязательно является последовательной. В подавляющем большинстве случаев вы будете иметь дело только с последовательностями терминов, поэтому мы будем в основном использовать слово «фраза» для простоты на протяжении всей книги, ссылаясь как на отдельные термины, так и на многочленные последовательности. Чтобы избежать путаницы, обратите внимание, что слово «термин» отдельно используется для обозначения «уникального значения в поле в поисковой системе». Таким образом, мы иногда также будем называть неразделенные строки с несколькими словами в поисковой системе «терминами», хотя с лингвистической точки зрения они считаются «фразами» или «последовательностями терминов».

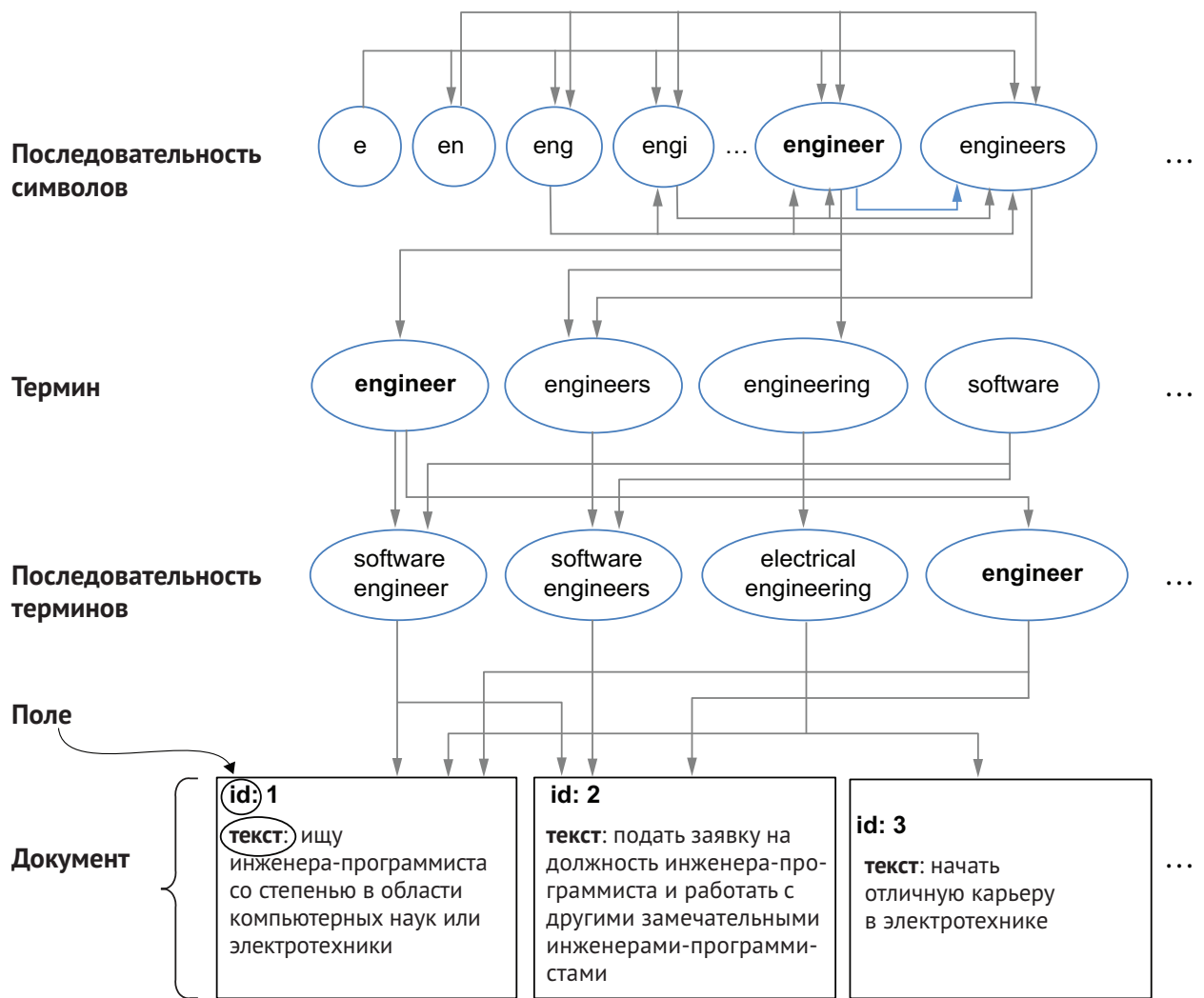


Рис. 2.8. Семантические данные, закодированные в свободном текстовом контенте. Символы образуют последовательности символов, которые образуют термины, которые образуют последовательности терминов, которые образуют поля, которые образуют документы, которые образуют корпус¹

Конечно, мы знаем, что несколько последовательностей терминов вместе могут образовывать предложения, несколько предложений могут образовывать абзацы, а абзацы затем могут быть свернуты в еще более крупные группы текста. Однако для поисковой системы следующим более высоким уровнем группировки, на котором мы обычно сосредоточимся после последовательностей терминов, является поле. *Поле* в поисковой системе – это разделенный и маркированный раздел документа, обычно для целей поиска или возврата в качестве независимой части документа. Поля, содержащие текст, можно анализировать любым количеством способов с помощью текстового анализатора, который обычно

¹ Корпус в программировании – это большой и структурированный набор текстов, используемых для лингвистического анализа и обработки естественного языка. Он может состоять из чего угодно: от новостных статей и книг до твитов и расшифровок разговорной речи. – *Прим. ред.*

включает такие методы, как разделение по пробелам и знакам препинания, перевод всех терминов в нижний регистр (lowercasing), чтобы быть нечувствительным к регистру, удаление шума (слов и определенных символов), стемминг¹ (stemming) или лемматизация² (lemmatization) для приведения терминов к базовой форме и удаление акцентов. Если процесс анализа текста вам незнаком или вы хотите освежить знания, мы рекомендуем ознакомиться с главой 6 книги «Solr в действии» Трея Грейнджера и Тимоти Поттера (Manning, 2014).

Затем одно или несколько полей объединяются в документ, а несколько документов образуют *корпус* (corpus) или коллекцию данных. Всякий раз, когда запрос выполняется по поисковому индексу, он фильтрует корпус в *набор документов*, являющийся подмножеством корпуса, которое конкретно относится к рассматриваемому запросу.

Каждый из этих лингвистических уровней – последовательности символов, термины, последовательности терминов, поля, документы, наборы документов и корпус – предоставляет уникальные идеи для понимания вашего контента и его уникального значения в вашей конкретной области.

2.3. Распределительная семантика и эмбединги

Распределительная семантика – это область исследований в сфере обработки естественного языка, которая фокусируется на семантических отношениях между терминами и фразами на основе распределительной гипотезы. Распределительная гипотеза заключается в том, что слова, которые встречаются в схожих контекстах, как правило, имеют схожие значения. Она хорошо резюмируется популярной цитатой: «Вы узнаете слово по компании, в которой его найдете»³.

При применении подходов машинного обучения к вашему тексту эта распределительная семантика становится все более важной, и поисковый движок делает невероятно простым получение контекста

¹ Стемминг – это процесс сокращения слова до его корневой или базовой формы. Это делается путем удаления суффиксов, префиксов и других инфиксов. Стемминг используется для группировки слов с одинаковым корнем, что помогает в таких задачах, как классификация документов, анализ настроения и поиск информации. – *Прим. ред.*

² Лемматизация – важнейший метод обработки естественного языка. Он играет важную роль в предварительной обработке текста, преобразуя слова в их базовые или корневые формы, известные как леммы. Этот процесс помогает стандартизировать слова, которые встречаются в различных грамматических формах, уменьшая сложность текстовых данных и повышая точность чтения. – *Прим. ред.*

³ Джон Руперт Фирт, «Синописис лингвистической теории, 1930–1955», в J. R. Firth et al., Исследования по лингвистическому анализу, Специальный том Филологического общества (Издательство Оксфордского университета, 1957).

для большинства языковых представлений, присутствующих в вашем корпусе. Например, если кто-то хочет найти все документы о руководителях высшего звена, он может выполнить такой запрос:

`c?o`

Этот запрос будет соответствовать «CEO», «CMO», «CFO» или любому другому названию в стиле СХО, поскольку он запрашивает любую последовательность символов, начинающуюся с «с» и заканчивающуюся на «о» с одним символом между ними.

Такая же свобода существует для запроса произвольно сложных последовательностей терминов:

`«VP Engineering»~2`

Этот запрос будет соответствовать «VP Engineering», «VP of Engineering», «Engineering VP» или даже «VP of Software Engineering», поскольку он запрашивает поиск «VP» и «Engineering» в пределах двух позиций (расстояний редактирования) друг от друга.

Конечно, природа инвертированного индекса поисковой системы также делает тривиальной поддержку произвольных булевых запросов. Например, если кто-то ищет термин «Word», но мы хотим убедиться, что все соответствующие документы также содержат термин «Microsoft» или «MS» где-то в документе, можно было бы выполнить следующий запрос:

`(Microsoft OR MS) AND Word`

Поисковые системы поддерживают произвольно сложные комбинации запросов для последовательностей символов, терминов и последовательностей терминов по всему корпусу, возвращая наборы документов, которые служат уникальным контекстом контента, соответствующего этому запросу. Например, если вы выполните запрос для `pizza`, возвращенные документы, скорее всего, будут ресторанами, а не компаниями по прокату автомобилей, а если вы выполните запрос для `machine learning`, вы, скорее всего, увидите вакансии для специалистов по данным или инженеров-программистов, чем для бухгалтеров, работников сферы общественного питания или фармацевтов. Это означает, что вы можете вывести сильную связь между «машинным обучением» и «программной инженерией» и слабую связь между «машинным обучением» и «работником сферы общественного питания». Если вы копнете глубже, вы также сможете увидеть, какие другие термины и фразы чаще всего встречаются в наборе документов машинного обучения относительно остальной части вашего корпуса, и, таким образом, лучше понять значение и использование фразы «машинное обучение». Мы рассмотрим практические примеры использования этих отношений в главе 5.

Знакомство с векторами

Базовое понимание векторных операций будет важно по мере продвижения по этой книге. Вектор – это список значений, описывающих некоторые атрибуты предмета. Например, если ваши предметы – это дома, у вас может быть список атрибутов, таких как *price*, *size* и *number of bedrooms*. Если у вас есть дом стоимостью 100 000 долл. с площадью 1000 кв. футов и двумя спальнями, это можно представить как вектор [100 000, 1000, 2]. Эти атрибуты (цена, размер и количество спален в этом примере) называются *измерениями* (*dimensions*), или *признаками*, или характеристиками¹ (*feature*), а определенный набор измерений называется векторным пространством. Вы можете представить любые другие предметы (например, другие дома, квартиры или жилища) в том же векторном пространстве, если вы можете присвоить им значения в измерениях векторного пространства.

Если мы рассмотрим другие векторы в том же векторном пространстве (например, дом [1000000, 9850, 12] и другой дом [120000, 1400, 3]), мы можем выполнить математические операции над векторами, чтобы изучить тенденции и сравнить векторы. Например, вы можете интуитивно посмотреть на эти три примера векторов и определить, что «цены на дома имеют тенденцию увеличиваться по мере увеличения количества комнат» или что «количество комнат имеет тенденцию увеличиваться по мере увеличения размера дома». Мы также можем выполнить вычисления подобия над векторами, чтобы определить, что дом за 120 000 долл. с 1400 кв. футами и тремя спальнями больше похож на дом за 100 000 долл. с 1000 кв. футами и двумя спальнями, чем на дом за 1 000 000 долл. с 9 850 кв. футами и 12 спальнями.

В последние годы гипотеза распределения применялась для создания семантических представлений терминов и последовательностей терминов с помощью того, что известно как *эмбеddинг*. Эмбеddинг – это набор координат в векторном пространстве, в которое мы преобразуем (или «встраиваем») понятие. Более конкретно, этот набор координат – это числовой вектор (список чисел), который предназначен для представления семантического значения ваших данных (текста, изображения, аудио, поведения или других модальностей данных). Эмбеddинги на основе текста могут представлять последовательности терминов любой длины, но при представлении отдельных слов или фраз мы называем эти *эмбеddингом слов*.

Последовательность терминов часто кодируется в эмбеddинг с уменьшенной размерностью, который можно сравнить с векторами для всех других эмбеddингов в корпусе, чтобы найти наиболее семантически связанные документы.

¹ Сленговое – «фича», от англ. *feature*, в программировании – это функциональная возможность или характеристика программного обеспечения. Это специально разработанная особенность или возможность программного продукта, предназначенная для выполнения определенной задачи или улучшения пользовательского опыта. – *Прим ред.*

Чтобы понять этот процесс, может быть полезно подумать о том, как работает поисковый движок «из коробки». Давайте представим, что для каждого термина существует вектор, который содержит значение (измерение) для каждого слова в вашем корпусе. Это может выглядеть примерно так, как показано на рис. 2.9.

Запрос	Все термины из индекса										
	apple	caffeine	cheese	coffee	drink	donut	food	juice	pizza	tea	water
latte	0	0	0	0	0	0	0	0	0	0	0
cappuccino	0	0	0	0	0	0	0	0	0	0	0
apple juice	1	0	0	0	0	0	0	1	0	0	0
cheese pizza	0	0	1	0	0	0	0	0	1	0	0
donut	0	0	0	0	0	1	0	0	0	0	0
soda	0	0	0	0	0	0	0	0	0	0	0
green tea	0	0	0	0	0	0	0	0	0	1	0
water	0	0	0	0	0	0	0	0	0	0	1
cheese bread sticks	0	0	1	0	0	0	0	0	0	0	0
cinnamon sticks	0	0	0	0	0	0	0	0	0	0	0

Рис. 2.9. Векторы с одним измерением на термин из инвертированного индекса. Каждый запрос слева сопоставляется с вектором справа, со значением 1 для любого термина из индекса, который также есть в запросе, и 0 для любого термина из индекса, которого в запросе нет

Рисунок 2.9 демонстрирует, как по умолчанию работает сопоставление документов в лексических поисковых системах. *Лексический поиск* – это поиск, в котором документы сопоставляются и ранжируются на основе степени, в которой они содержат фактические ключевые слова или другие атрибуты, указанные в запросе. Для каждого запроса по ключевому слову существует вектор, который содержит измерение для каждого термина в инвертированном индексе. Если этот термин существует в запросе, значение в векторе равно 1 для этого измерения, а если этого значения в запросе нет, то значение равно 0 для этого измерения. Похожий вектор существует для каждого документа в инвертированном индексе, со значением 1 для любого термина из индекса, который появляется в документе, и 0 для всех других терминов.

Когда выполняется запрос, в индексе происходит поиск любых соответствующих терминов, а затем вычисляется оценка сходства на основе сравнения вектора для запроса и вектора для документа, который оценивается относительно запроса. Мы рассмотрим конкретный расчет рейтинга в главе 3, но этого общего понимания на данный момент достаточно.

У этого подхода есть очевидные недостатки. Хотя он отлично подходит для поиска документов с точными совпадениями ключевых слов, что происходит, когда вы хотите найти «связанные» вещи? Например, вы заметите, что термин «газировка» появляется в запросе,

но никогда в индексе. Несмотря на то что есть другие виды напитков («яблочный сок», «вода», «капучино» и «латте»), поисковый движок всегда будет возвращать нулевые результаты, потому что он не понимает, что пользователь ищет напиток. Аналогично вы заметите, что даже несмотря на то, что термин «кофеин» существует в индексе, запросы по латте, капучино и зеленому чаю никогда не будут соответствовать термину «кофеин», даже если они связаны.

По этим причинам в настоящее время общепринятой практикой является использование плотных эмбедингов с уменьшенной размерностью для моделирования семантического значения последовательностей терминов в вашем индексе и запросах. *Плотный эмбединг* (также известный как *эмбединг плотных векторов*)¹ – это вектор более абстрактных признаков, который кодирует значение ввода в семантическом пространстве. Рисунок 2.10 демонстрирует термины, теперь сопоставленные с вектором с уменьшенной размерностью, который может служить плотным эмбедингом.

	food	drink	dairy	bread	caffeine	sweet	calories	healthy
apple juice	0	5	0	0	0	4	4	3
cappuccino	0	5	3	0	4	1	2	3
cheese bread sticks	5	0	4	5	0	1	4	2
cheese pizza	5	0	4	4	0	1	5	2
cinnamon bread sticks	5	0	1	5	0	3	4	2
donut	5	0	1	5	0	4	5	1
green tea	0	5	0	0	2	1	1	5
latte	0	5	4	0	4	1	3	3
soda	0	5	0	0	3	5	5	0
water	0	5	0	0	0	0	0	5

Рис. 2.10. Плотные эмбединги с уменьшенной размерностью. В этом случае вместо одного измерения на термин (существует или отсутствует) теперь существуют измерения более высокого уровня, которые оценивают общие атрибуты для предметов, такие как «здоровый», содержит «кофеин», «хлеб» или «молочные продукты» или является ли предмет «едой» или «напитком»

С новым вектором эмбединга, который теперь доступен для каждой последовательности терминов в крайнем левом столбце рис. 2.10, мы теперь можем оценить связь между каждой парой последовательностей терминов, используя сходство между их векторами. В линейной алгебре мы используем функцию косинусного сходства (или другую меру сходства) для оценки связи между двумя векторами. Ко-

¹ Эмбединг плотных векторов в программировании – это числовое представление данных, которое позволяет моделям анализировать и интерпретировать текст. Он заключается в преобразовании слов, предложений или их частей в многомерные векторы, где каждое измерение отражает характеристику или связь с другими элементами текста. – *Прим. ред.*

синусное сходство вычисляется путем выполнения скалярного произведения между двумя векторами и масштабирования его по величинам (длинам) каждого из векторов. Мы рассмотрим математику более подробно в следующей главе, но сейчас рис. 2.11 показывает результаты оценки сходства между несколькими из этих векторов.

Term Sequence:	Vector:
apple juice:	[1, 5, 0, 0, 0, 4, 4, 3]
cappuccino:	[0, 5, 3, 0, 4, 1, 2, 3]
cheese bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
cheese pizza:	[5, 0, 4, 4, 0, 1, 5, 2]
cinnamon bread sticks:	[5, 0, 4, 5, 0, 1, 4, 2]
donut:	[5, 0, 1, 5, 0, 4, 5, 1]
green tea:	[0, 5, 0, 0, 2, 1, 1, 5]
latte:	[0, 5, 4, 0, 4, 1, 3, 3]
soda:	[0, 5, 0, 0, 3, 5, 5, 0]
water:	[0, 5, 0, 0, 0, 0, 0, 5]

Векторное сходство (a, b):

$$\cos(\theta) = \frac{a \cdot b}{|a| \times |b|}$$

Оценки векторного сходства:

Green Tea	
0.94	water
0.85	cappuccino
0.80	latte
0.78	apple juice
0.60	soda
...	...
0.19	donut

Оценки векторного сходства:

Cheese Pizza	
0.99	cheese bread sticks
0.91	cinnamon bread sticks
0.89	donut
0.47	latte
0.46	apple juice
...	...
0.19	water

Оценки векторного сходства:

Donut	
0.99	cinnamon bread sticks
0.89	cheese bread sticks
0.89	cheese pizza
0.57	apple juice
0.51	soda
...	...
0.07	water

Рис. 2.11. Сходство между эмбедингами. Косинус между векторами показывает список предметов, отсортированный по сходству с «зеленым чаем», с «сырной пиццей» и с «пончиком»

Как вы можете видеть на рис. 2.11, поскольку каждая последовательность терминов закодирована в вектор, который представляет ее значение в терминах признаков более высокого уровня, этот эмбединг теперь можно использовать для оценки сходства этой последовательности терминов с любым другим похожим вектором. Вы увидите три списка сходства векторов в нижней части рисунка: один для «зеленого чая», один для «пиццы с сыром» и один для «пончика».

Сравнивая векторное сходство «зеленого чая» со всеми другими последовательностями терминов, мы обнаруживаем, что наиболее связанными предметами являются «вода», «капучино», «латте», «яблочный сок» и «газировка», а наименее связанным является «пончик». Это интуитивно понятно, так как «зеленый чай» имеет больше общих атрибутов с предметами, расположенными выше в списке. Для вектора «пицца с сыром» мы видим, что наиболее похожие другие эмбединги – для «палочек сыра», «палочек корицы» и «пончика», а «вода» находится в конце списка. Наконец, для термина «пончик» мы обнаруживаем, что наиболее похожими предметами являются «палочки корицы», «палочки сыра» и «пицца с сыром», а «вода» снова находится в конце списка. Эти результаты отлично справляются с поиском наиболее похожих предметов на наш исходный запрос.

Стоит отметить, что эта векторная оценка используется только при расчете сходства между предметами. В вашей поисковой системе обычно есть двухфазный процесс, при котором вы сначала фильтруете набор документов (фаза сопоставления), а затем оцениваете эти полученные документы (фаза ранжирования). Если вы не собираетесь пропустить первый шаг и оценить все ваши документы относительно векторов запроса (что может быть трудоемким и требующим больших затрат времени и обработки), вам все равно понадобится некоторая форма начального сопоставления до фазы ранжирования, чтобы отфильтровать запрос по разумному количеству документов для оценки. Мы подробнее рассмотрим механизмы успешной реализации эмбедингов и поиска векторов в главах 3, 9, 13, 14 и 15.

Эмбединги могут представлять запросы, части документов или даже целые документы. Обычно термины и последовательности терминов кодируются в эмбединги слов, но *эмбединги предложений* (кодирование вектора со значением предложения), *эмбединги абзацев* (кодирование вектора со значением абзаца) и *эмбединги документов* (кодирование вектора со значением всего документа) также являются распространенными методами.

Также очень часто измерения бывают более абстрактными, чем наши примеры здесь. Например, модели глубокого обучения, такие как LLM, могут обнаруживать, казалось бы, непонятные признаки в последовательностях символов и того, как документы группируются вместе в корпусе. Мы не смогли бы легко применить понятную человеку метку к этим измерениям в векторе эмбединга, но пока это улучшает предсказательную силу модели и увеличивает релевантность, это обычно не является проблемой для большинства поисковых команд. Фактически, поскольку векторы кодируют «смысл» с помощью различных абстрактных числовых признаков, также возможно создавать и выполнять поиск по векторам, представляющим различные типы (или модальности) данных, такие как изображения, аудио, видео или даже сигналы и шаблоны активности. Мы рассмотрим мультимодальный поиск (поиск по различным модальностям данных) в разделе 15.3.

В конечном счете объединение нескольких моделей для использования возможностей распределительной семантики и эмбединга, как правило, приводит к наилучшим результатам, и мы более подробно рассмотрим многочисленные графические и векторные подходы к использованию этих методов в оставшейся части этой книги.

2.4. Моделирование домен-специфичных знаний

В главе 1 мы обсудили прогрессию поискового интеллекта (см. рис. 1.8), в соответствии с которым организации начинают с базового поиска по ключевым словам и проходят несколько дополнительных этапов совершенствования, прежде чем они в конечном итоге достигнут полной самообучающейся системы. Вторым этапом в этом развитии поискового интеллекта было построение таксо-

номий и онтологий, а третий этап («намерение запроса») включал построение и использование графов знаний. К сожалению, иногда среди практиков в отрасли может возникать значительная путаница в отношении правильных определений и ключевых терминов, таких как «онтология», «таксономия», «списки синонимов», «графы знаний», «альтернативные метки» и т. д. Нам будет полезно предоставить некоторые определения для использования в этой книге, чтобы избежать любой двусмысленности. В частности, мы изложим определения для ключевых терминов «граф знаний», «онтология», «таксономия», «синонимы» и «альтернативные метки». На рис. 2.12 показана высокоуровневая схема того, как они связаны.

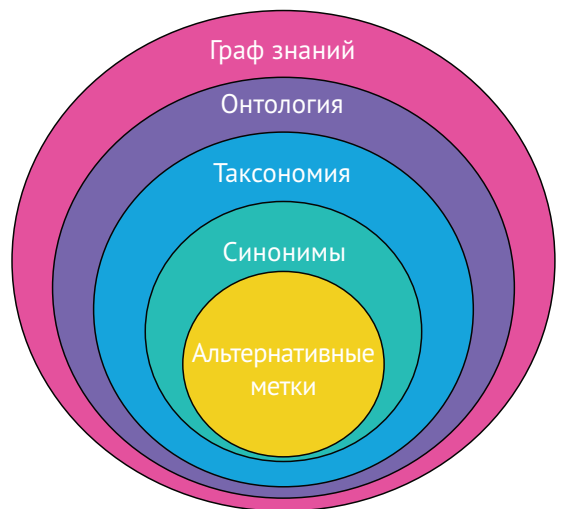


Рис. 2.12. Уровни моделирования предметно-ориентированных знаний. Графы знаний расширяют онтологии, которые расширяют таксономии. Синонимы расширяют альтернативные метки и сопоставляются с входами в таксономии

Мы определяем каждый из этих методов моделирования знаний следующим образом:

- *альтернативные метки* (alt. labels) – последовательности терминов замены с идентичными значениями:

Examples:

CTO => Chief Technology Officer
specialise => specialize

- *синонимы* – последовательности терминов замены, которые могут использоваться для представления одинаковых или очень похожих вещей:

Examples:

human => homo sapiens, mankind
food => sustenance, meal

- *таксономия* – классификация вещей по категориям:

Examples:

human is mammal
mammal is animal

- *онтология* – отображение отношений между типами вещей:

Examples:

animal eats food
food contains ingredients

- *граф знаний* – экземпляр онтологии, который также содержит связанные вещи:

Examples:

John is human

John eats food

Создание альтернативных меток – самый простой из этих методов для понимания. Инициализмы (например, «RN» => «Registered Nurse») и аббревиатуры практически всегда служат альтернативными метками, как и опечатки и альтернативные варианты написания. Иногда полезно хранить эти сопоставления в отдельных списках, особенно если вы используете алгоритмы для их определения и ожидаете, что сможете вносить в них изменения человеком, или если вы планируете повторно запустить алгоритмы позже.

Синонимы – это следующий по распространенности метод, поскольку практически каждая поисковая система будет иметь некоторую реализацию списка синонимов. Альтернативные метки – это подмножество списка синонимов, и они являются наиболее очевидным видом синонимов. Большинство людей также считают тесно связанные последовательности терминов синонимами. Например, «программный инженер» и «разработчик программного обеспечения» часто считаются синонимами, поскольку они обычно используются взаимозаменяемо, хотя между ними есть некоторые нюансы в значении. Иногда вы даже увидите переводы слов между языками, отображаемые в синонимах для двуязычных вариантов использования поиска.

Одним из ключевых различий между альтернативными метками и более общими синонимами является то, что альтернативные метки можно рассматривать как *термины замены* для оригинала, тогда как синонимы чаще используются как *термины расширения* для добавления к оригиналу. Реализации могут сильно различаться, но в конечном итоге это сводится к тому, уверены ли вы, что две последовательности терминов имеют одинаковое значение (и вы хотите их нормировать), или вы просто пытаетесь включить дополнительные связанные последовательности терминов, чтобы не пропустить другие релевантные результаты.

Таксономии – это следующий шаг после синонимов. Таксономии меньше фокусируются на словах замены или расширения, а вместо этого фокусируются на категоризации вашего контента в иерархию. Таксономическая информация часто будет использоваться для управления навигацией по веб-сайту, для изменения поведения подмножества результатов поиска (например, для отображения различных вариантов фасетирования или фильтрации на основе родительской категории продукта) или для применения динамической фильтрации на основе категории, с которой сопоставляется запрос. Например, если кто-то ищет «диапазон» на веб-сайте по ремонту дома, сайт может автоматически отфильтровать до «бытовой техники», чтобы удалить шум других продуктов, которые содержат фразы типа «попадают в ди-

апазон» в своем описании продукта. Синонимы затем сопоставляются с таксономией, указывая на определенные элементы в таксономии.

В то время как таксономии, как правило, определяют родительско-дочерние отношения между категориями, а затем сопоставляют вещи с этими категориями, онтологии предоставляют возможность определять гораздо более богатые отношения между вещами (последовательности терминов, сущности) в пределах домена. Онтологии обычно определяют более абстрактные отношения, пытаясь смоделировать отношения между видами вещей в домене, такими как «сотрудник подчиняется начальнику», «начальник директора по маркетингу является генеральным директором», «директор по маркетингу является сотрудником». Это делает онтологии действительно полезными для получения новой информации из известных фактов путем сопоставления фактов с онтологией и последующих логических выводов на основе отношений в онтологии, которые можно применить к этим фактам.

Графы знаний являются относительной новостью в области управления знаниями. В то время как онтологии определяют высокоуровневые отношения, которые применяются к типам вещей, графы знаний, как правило, являются полными экземплярами онтологий, которые также включают каждую из конкретных сущностей, попадающих в эти типы. Используя наш предыдущий пример онтологии, граф знаний дополнительно имел бы «Майкл – директор по маркетингу», «Майкл подчиняется Марсии» и «Марсия – генеральный директор» в качестве отношений в графе. До того как графы знаний вышли на передний план, было принято моделировать эти более подробные отношения в онтологии, и многие люди все еще делают это сегодня. В результате вы часто будете видеть, что термины «граф знаний» и «онтология» используются взаимозаменяемо, хотя со временем это становится все менее распространенным.

В этой книге мы в основном сосредоточимся на обсуждениях альтернативных меток, синонимов и графов знаний, поскольку таксономии и онтологии в основном включены в графы знаний. Более подробно мы рассмотрим графы знаний в главе 5.

2.5. Проблемы понимания естественного языка для поиска

В последних нескольких разделах мы обсудили богатый граф смысла, встроенный в неструктурированные данные, такие как текст, а также то, как распределительная семантика и эмбединг могут использоваться для вывода и оценки семантических отношений между последовательностями терминов в запросах и документах. Мы также представили ключевые методы моделирования знаний и определили связанную терминологию, которую будем использовать в этой книге. В этом разделе мы обсудим несколько ключевых проблем, связанных с пониманием естественного языка, которые постараемся преодолеть в следующих главах.

2.5.1. Проблема неоднозначности (полисемия)

В разделе 2.1.3 мы ввели понятие полисемии, или неоднозначных терминов. В этом разделе мы имели дело с изображением, помеченным именем «Тре́й Грейнджер», но которое относилось к другому человеку, нежели к автору этой книги. Однако в текстовых данных у нас та же проблема, и она может стать очень запутанной.

Рассмотрим такое слово, как «driver». Оно может в широком смысле относиться к водителю транспортного средства, к разновидности клюшки для гольфа для удара по мячу, к программному обеспечению, которое позволяет аппаратному устройству работать, к разновидности инструмента (отвертка) или к стимулу для продвижения чего-либо вперед («ключевой фактор успеха»). У этого слова есть много потенциальных значений, и вы можете изучить еще более подробные значения. Например, в категории «водитель транспортного средства» это может означать водителя такси, водителя Uber, водителя Lyft, профессионального дальнобойщика, например водителя CDL (человека с коммерческими водительскими правами) или даже водителя автобуса. В подгруппе водителей автобусов это может означать водителя школьного автобуса, водителя общественного городского автобуса, водителя туристического автобуса и т. д. Мы могли бы продолжить разбивать этот список на десятки дополнительных категорий как минимум.

При создании поисковых приложений инженеры часто наивно создают статические списки синонимов и предполагают, что термины имеют единственное значение, которое может применяться универсально. Однако реальность такова, что каждый термин (слово или фраза) приобретает уникальное значение, основанное на конкретном контексте, в котором он используется.

СОВЕТ. Каждый термин приобретает уникальное значение, основанное на конкретном контексте, в котором он используется.

Не всегда практично поддерживать бесконечное количество потенциальных значений, хотя мы обсуждаем методы приближения этого с помощью семантического графа знаний в главе 5. Тем не менее независимо от того, поддерживаете ли вы много значений для каждой фразы или только несколько, важно осознавать очевидную необходимость иметь возможность генерировать точную (и часто очень тонкую) интерпретацию для любой заданной фразы, с которой могут столкнуться ваши пользователи.

2.5.2. Проблема понимания контекста

Я люблю говорить, что каждый термин (слово или фраза), с которым вы когда-либо сталкиваетесь, представляет собой «зависимый от контекста кластер значений с неоднозначной меткой». То есть имеется метка (текстовое представление термина), которая применяется к некоторому понятию (кластеру значений) и зависит от контекста, в котором она находится. Согласно этому определению невозможно точно интерпретиро-

вать термин, не понимая его контекста. В результате создание фиксированных списков синонимов, которые не могут учитывать контекст, скорее всего, создаст неоптимальный опыт поиска для ваших пользователей.

Модели трансформеров в основном работают на этой предпосылке, используя подсказки ввода в качестве контекста, в котором интерпретируется каждая часть слова (или токен) в подсказке. Внимание уделяется каждому токenu на основе окружающих токенов и того, как они соотносятся с изученным представлением в модели, что также является контекстным. Мы углубимся в нюансы работы трансформеров в главе 13 и настроим трансформер для задачи ответа на вопрос в главе 14.

То, что контекст важен, не означает, что его всегда легко применить правильно. Часто необходимо выполнять базовый поиск ключевых слов в качестве запасного варианта, когда ваша поисковая система не понимает запрос, и почти всегда полезно иметь предварительно созданное понимание домена, на которое можно также положиться, чтобы помочь интерпретировать запросы. Это предварительно созданное понимание домена затем в конечном итоге переопределяет некоторые из стандартных действий сопоставления на основе ключевых слов (например, объединение отдельных ключевых слов во фразы, введение синонимов и исправление опечаток). Как мы обсуждали в главе 1, контекст запроса включает в себя не только ключевые слова поиска и содержимое ваших документов. Он также включает в себя понимание вашего домена, а также понимание вашего пользователя. Запросы могут приобретать совершенно разные значения в зависимости от того, что вы знаете о своем пользователе, и любого понимания, специфичного для домена, которое у вас может быть. Этот контекст необходим как для обнаружения, так и для разрешения видов неоднозначности, которые мы обсуждали в последнем разделе, а также для того, чтобы гарантировать, что ваши пользователи получают максимально возможный интеллектуальный опыт поиска. В этой книге мы сосредоточимся на методах автоматического изучения контекстных интерпретаций каждого запроса на основе уникального контекста, в котором он используется.

2.5.3. Проблема персонализации

При рассмотрении пользовательского контекста как инструмента для улучшения понимания запроса не всегда очевидно, как применить пользовательскую персонализацию поверх уже существующего контента и оценки, специфичной для домена. Например, предположим, вы узнали, что определенному пользователю действительно нравится Apple как бренд, потому что он продолжает искать iPhone. Означает ли это, что также следует делать бустинг Apple, когда он ищет часы, компьютеры, клавиатуры, наушники и музыкальные плееры? Возможно, пользователю нравятся только телефоны марки Apple, и, продвигая бренд в других категориях, вы можете разочаровать пользователя. Например, даже если пользователь ранее искал iPhone, как вы можете быть уверены, что он не просто пытался сравнить iPhone с другими рассматриваемыми телефонами?

Из всех измерений намерения пользователя (рис. 1.5) персонализация – это то, на чем легче всего споткнуться, и, как следствие, она реже всего встречается в современных поисковых приложениях на базе ИИ (конечно, за исключением рекомендательных систем). Мы подробно рассмотрим эти проблемы в главе 9, чтобы подчеркнуть, как мы можем найти правильный баланс при развертывании персонализированного поиска.

2.5.4. Проблемы интерпретации запросов по сравнению с интерпретацией документов

Одной из распространенных проблем, с которой мы сталкиваемся, когда инженеры и специалисты по данным впервые начинают работать с поиском, является склонность применять к запросам стандартные методы обработки естественного языка, такие как определение языка, определение части речи, определение фраз и анализ настроений. Обычно эти методы были обучены работать с более длинными блоками текста – часто на уровне документа, абзаца или по крайней мере предложения.

Документы, как правило, длиннее и предоставляют значительно больше контекста окружающему тексту, тогда как запросы – короткие (только несколько ключевых слов) в большинстве случаев использования. Даже если они длиннее, запросы по обыкновению объединяют несколько идей, а не предоставляют большое количество лингвистического контекста. В результате при попытке интерпретировать запросы вам нужно использовать внешний контекст как можно больше.

Вместо того чтобы использовать библиотеку обработки естественного языка, которая полагается на структуру предложений для интерпретации запроса, например, вы можете попробовать найти фразы из вашего запроса в корпусе документов, чтобы найти их наиболее распространенные домен-специфичные интерпретации. Аналогичным образом вы можете использовать совместное появление терминов в вашем запросе в предыдущих сеансах поиска пользователей, извлекая сигналы поведения пользователей. Это позволяет вам узнать реальные намерения похожих пользователей, которые было бы очень сложно надежно вывести из стандартной библиотеки обработки естественного языка.

Короче говоря, запросы требуют специальной обработки и интерпретации из-за их тенденции быть короткими, часто они подразумевают больше, чем явно заявляют, поэтому полное использование методов науки о данных для запросов, ориентированных на поиск, даст гораздо лучшие результаты, чем традиционные подходы к обработке естественного языка.

2.5.5. Проблемы интерпретации намерения запроса

Хотя процесс разбора запроса для понимания содержащихся в нем терминов и фраз важен, за запросом часто стоит намерение более высокого уровня – намерение запроса, если хотите. Например, давайте рассмотрим неотъемлемые различия между следующими запросами:

who is the CEO?
support
iphone screen blacked out
iphone
verizon silver iphone 8 plus 64GB
sale
refrigerators
pay my bill

Цель первого запроса *who is the CEO?* – явно найти фактический ответ, а не список документов. Второй запрос *support* пытается перейти в раздел *support* на веб-сайте или иным образом связаться со службой поддержки. Третий запрос *iphone screen blacked out* также ищет поддержку, но он касается конкретной проблемы: человек, вероятно, хочет найти страницы устранения неполадок, которые могут помочь с этой конкретной проблемой, прежде чем обращаться в реальную службу поддержки.

Следующие два запроса *iphone* и *verizon silver iphone 8 plus 64GB* довольно интересны. Хотя они оба для iPhone, первый поиск – это общий поиск, вероятно, указывающий на намерение просмотра или изучения продукта, тогда как второй запрос – гораздо более конкретный вариант первого поиска, указывающий на то, что пользователь точно знает, что он ищет, и может быть ближе к принятию решения о покупке. Общий запрос *iphone* может быть лучше для возврата целевой страницы, предоставляющей обзор iPhone и доступных вариантов, тогда как более конкретный запрос может быть лучше для перехода прямо на страницу продукта с кнопкой покупки. Как правило, чем более общий запрос, тем более вероятно, что пользователь просто просматривает. Более конкретные запросы, особенно когда они ссылаются на конкретные продукты по названию, часто указывают на намерение покупки или желание найти определенный известный продукт.

Запрос на продажу *sale* указывает, что пользователь ищет продукты, доступные для покупки по сниженной цене, что вызовет некий специально реализованный фильтр или перенаправит на определенную целевую страницу для текущего события распродажи. Запрос на *refrigerators* указывает, что пользователь хочет просмотреть определенную категорию документов о продуктах. Наконец, запрос на оплату счета *pay my bill* указывает, что пользователь хочет выполнить действие, – лучшим ответом на этот запрос является не набор результатов поиска или даже ответ, а вместо этого перенаправление на раздел просмотра и оплаты счетов приложения.

Каждый из этих запросов содержит намерение, выходящее за рамки простого набора ключевых слов для сопоставления. Независимо от того, является ли намерение перенаправлением на определенную страницу, применением определенных фильтров, просмотром или покупкой продуктов или даже выполнением действий, специфичных для домена, суть в том, что существуют нюансы, специфичные для домена, и в том, как пользователи могут выражать свои цели вашей по-

исковой системе. Часто бывает сложно автоматически вывести эти домен-специфичные намерения пользователя. Довольно часто компании внедряют определенные бизнес-правила для обработки этих одноразовых запросов. Классификаторы намерений запросов, безусловно, могут быть созданы для обработки подмножеств этой задачи, но успешная интерпретация всех возможных намерений запросов по-прежнему остается сложной задачей при построении возможностей интерпретации запросов на естественном языке.

2.6. Контент + сигналы: топливо, питающее поиск на основе ИИ

В первой главе мы представили идею отраженного интеллекта – использования циклов обратной связи для постоянного обучения как на основе контента, так и на основе взаимодействий пользователей. Эта глава была полностью сосредоточена на понимании смысла и интеллекта, встроенных в ваш контент, но важно признать, что многие из методов, которые мы применим к неструктурированным данным в ваших документах, также могут быть легко применены к вашим поведенческим сигналам пользователя. Например, в разделе 2.3 мы обсуждали, как значения фраз могут быть получены путем поиска других фраз, с которыми они чаще всего встречаются в вашем корпусе. Мы отметили, что «машинное обучение» чаще встречается со «специалистом по данным» и «инженером-программистом», чем с «бухгалтерами», «работниками сферы общественного питания» или «фармацевтами». Если вы абстрагируете гипотезу распределения от документов и также примените ее к поведению пользователей, вы можете ожидать, что похожие пользователи, запрашивающие вашу поисковую систему, вероятно, будут демонстрировать похожее поведение запросов. В частности, люди, которые являются специалистами по данным или ищут специалистов по данным, гораздо более склонны также искать или взаимодействовать с документами о машинном обучении, и вероятность того, что работник сферы общественного питания или бухгалтер будет искать контент о машинном обучении, намного ниже, чем вероятность того, что это сделает инженер-программист. Таким образом, мы можем применять эти же методы для изучения связанных терминов и последовательностей терминов из журналов запросов, где вместо того, чтобы думать о терминах и последовательностях терминов, сопоставляемых с полями в документах, мы думаем о терминах в запросах и кликах по результатам поиска, сопоставляемых с сеансами пользователей, которые затем сопоставляются с пользователями. Мы будем следовать этому подходу в главе 6, чтобы узнать связанные термины, синонимы и орфографические ошибки из журналов запросов пользователей.

Некоторые поисковые приложения богаты контентом, но имеют очень мало пользовательских сигналов. Другие поисковые при-

ложения имеют огромное количество сигналов, но либо очень мало контента, либо контент, который создает проблемы с точки зрения автоматизированного обучения. В идеальном сценарии у вас будет отличный контент и огромное количество пользовательских сигналов для обучения, что позволит вам объединить лучшее из того и другого в еще более умное поисковое приложение на базе ИИ. Независимо от того, в каком сценарии вы находитесь, помните, что ваш контент и ваши пользовательские сигналы могут служить топливом для работы алгоритмов обучения, и вы должны сделать все возможное, чтобы максимизировать сбор и качество каждого из них.

В качестве последнего замечания о понимании естественного языка: с появлением LLM, которые представляют собой глубокие нейронные сети, обученные на большом проценте человеческих знаний (большая часть интернета, а также выбранные источники), у нас теперь есть возможность интерпретировать значение и намерение вопросов общего знания на беспрецедентном уровне качества. LLM не очень хорошо справляются с пониманием предметной области «из коробки», по крайней мере, для информации, которая не является частью их обучающих наборов, но с возможностью тонкой настройки LLM на предметно-специфических данных эти модели часто можно быстро адаптировать к более закрытым данным предметной области. LLM представляют собой большой скачок вперед в нашей способности изучать нюансы естественного языка, интерпретировать произвольные документы и запросы на основе этих нюансов и осуществлять более релевантный поиск.

LLM, хотя в целом являются наиболее впечатляющей техникой для широкомасштабного понимания естественного языка, далеко не единственные мощные инструменты в нашем наборе инструментов поиска на основе ИИ. Мы углубимся в использование LLM для поиска в главах 9, 13, 14 и 15. В то же время у нас есть много других важных алгоритмов и методов для изучения понимания естественного языка и предметной области, интерпретации поведения пользователя и обучения оптимальным моделям ранжирования релевантности. Теперь, когда мы рассмотрели всю необходимую информацию для извлечения смысла из контента на естественном языке, пришло время засучить рукава и приступить к делу. В следующей главе мы рассмотрим множество примеров, поскольку начнем изучать релевантность на основе контента в поисковом приложении на базе ИИ.

Резюме

- Неструктурированные данные – это неправильное название, на самом деле они больше похожи на гиперструктурированные данные, поскольку представляют собой гигантский граф знаний, специфичных для предметной области.
- Поисковые системы могут использовать распределительную семантику – интерпретируя семантические отношения между тер-

минами и фразами на основе распределительной гипотезы – для использования богатого семантического значения на уровне последовательностей символов, терминов, последовательностей терминов (обычно фраз), полей, документов, наборов документов и всего корпуса.

- Подходы распределительной семантики позволяют нам узнавать нюансы смысла наших запросов и контента из их более широкого окружающего контекста.
- Эмбединг – это мощный метод ранжирования результатов поиска на основе семантического значения текста (и других модальностей данных), а не только на наличии и встречаемости определенных ключевых слов.
- Знания, специфичные для предметной области, обычно моделируются посредством комбинации альтернативных меток, списков синонимов, таксономий, онтологий и графов знаний. Графы знаний обычно моделируют вывод каждого из других подходов в единое представление знаний определенной предметной области.
- Полисемия (неоднозначные термины), контекст, персонализация и подходы к обработке естественного языка, специфичные для запроса, представляют собой некоторые из наиболее интересных задач в поиске на естественном языке.
- Контент и пользовательские сигналы являются важным топливом для наших поисковых приложений на базе ИИ, которые используются при решении задач на естественном языке.

3

Ранжирование и релевантность на основе контента

https://t.me/it_boooks/2

В этой главе рассматривается:

- выполнение запросов и возврат результатов поиска;
- ранжирование результатов поиска на основе их релевантности входящему запросу;
- соответствие ключевых слов и фильтрация по сравнению с векторным ранжированием;
- управление и указание пользовательских функций ранжирования с помощью запросов функций;
- обслуживание функций ранжирования для определенного домена.

Поисковые системы в основном выполняют три действия: принимают контент (*индексирование*), возвращают контент, соответствующий входящим запросам (*сопоставление*), и сортируют возвращенный контент на основе некоторой меры того, насколько хорошо он соответствует запросу (*ранжирование*). Могут быть добавлены дополнительные слои, позволяющие пользователям предоставлять лучшие запросы (автоподсказки, диалоги чат-бота и т. д.) и извлекать лучшие ответы из результатов или

суммировать результаты с помощью больших языковых моделей (см. главы 14–15), но основными функциями поисковой системы являются сопоставление и ранжирование на основе индексированных данных.

Релевантность – это понятие того, насколько хорошо возвращаемый контент соответствует запросу. Обычно сопоставляемый контент – это документы, а возвращаемый и ранжируемый контент – это сопоставленные документы вместе с соответствующими метаданными. В большинстве поисковых систем сортировка по релевантности по умолчанию основана на рейтинге, указывающем, насколько хорошо каждое ключевое слово в запросе соответствует тому же ключевому слову в каждом сопоставленном документе. В качестве альтернативы запросы можно сопоставить с числовыми векторными представлениями, а затем оценка будет представлять, насколько похож вектор запроса на каждый сопоставленный документ. Лучшие соответствия дают наивысший рейтинг релевантности, и они возвращаются в верхней части результатов поиска. Расчет релевантности легко настраивается и может быть скорректирован для каждого запроса, что позволяет реализовать сложное поведение ранжирования.

В этой главе мы дадим обзор того, как рассчитывается релевантность, как можно легко контролировать и корректировать функцию релевантности с помощью функциональных запросов и как мы можем реализовать популярные функции ранжирования релевантности, специфичные для домена и пользователя.

3.1. Оценка векторов запросов и документов с помощью косинусного сходства

В разделе 2.3 мы продемонстрировали идею измерения сходства двух векторов путем вычисления косинуса между ними. Мы создали векторы (списки чисел, где каждое число представляет силу некоторого признака), которые представляли различные продукты питания, а затем вычислили косинус угла между векторами, чтобы определить их сходство. Мы подробнее рассмотрим эту технику в текущем разделе ниже, обсудив, как текстовые запросы и документы могут отображаться в векторы (маппинг на векторы) для целей ранжирования. Затем мы рассмотрим некоторые популярные методы взвешивания признаков на основе текста и то, как их можно интегрировать для создания улучшенной формулы ранжирования релевантности.

Запуск примеров кода

Все листинги кода в книге доступны в блокнотах Jupyter, работающих в предварительно настроенных контейнерах Docker. Это позволяет вам запускать интерактивные версии кода с помощью одной команды (`docker compose up`), не тратя время на сложную настройку системы и управление зависимостями. Примеры кода также могут работать с несколькими поисковыми системами и векторными базами данных.

Инструкции по настройке и запуску блокнотов Jupyter и их использованию в веб-браузере см. в приложении А.

Для краткости в листингах этой книги могут отсутствовать некоторые строки кода, такие как импорт или вспомогательный код, но блокноты содержат все детали реализации.

В этом разделе мы рассмотрим первые листинги кода книги. Будет полезно запустить контейнеры Docker, необходимые для запуска сопутствующих блокнотов Jupyter, чтобы вы могли работать с интерактивными примерами кода. Инструкции по выполнению этого приведены в приложении А.

3.1.1. Преобразование текста в векторы

В типичном поисковом приложении мы начинаем с коллекции документов, а затем пытаемся ранжировать документы на основе того, насколько хорошо они соответствуют запросу пользователя. В этом разделе мы рассмотрим процесс маппинга (преобразования) текста запросов и документов в векторы¹.

В предыдущей главе мы использовали пример поиска продуктов питания и напитков, таких как яблочный сок, поэтому давайте воспользуемся этим примером здесь. Предположим, у нас есть два разных документа, которые мы хотели бы отсортировать на основе того, насколько хорошо они соответствуют этому запросу.

Запрос: apple juice

Document 1:

Lynn: ham and cheese sandwich, chocolate cookie, ice water Brian: turkey avocado sandwich, plain potato chips, apple juice Mohammed: grilled chicken salad, fruit cup, lemonade

Document 2:

Orchard Farms apple juice is premium, organic apple juice made from the freshest apples, never from concentrate. Its juice has received the regional award for best apple juice three years in a row.

Если мы сделаем маппинг обоих документов (содержащих в общей сложности 48 слов) в векторы, они отобразятся на векторное пространство из 48 слов со следующими размерами:

[a, and, apple, apples, avocado, award, best, brian, cheese, chicken, chips, chocolate, concentrate, cookie, cup, farms, for, freshest, from, fruit, grilled, ham, has, ice, in, is, its, juice, lemonade, lynn, made, mohammed, never, orchard, organic, plain, potato, premium, received, regional, row, salad, sandwich, the, three, turkey, water, years]

Если вы помните, в разделе 2.3 мы предложили рассматривать запрос для фразы «яблочный сок» как вектор, содержащий признак для

¹ Маппинг текста в векторы, англ. *mapping the text into vectors*, – это процесс преобразования текста в числовой формат, понятный для алгоритмов машинного обучения. По сути, текст превращается в векторы – наборы чисел, которые отражают определенные характеристики текста. – *Прим. ред.*

Листинг 3.1. Вычисление косинусного сходства между векторами

```

query_vector = numpy.array(
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

doc1_vector = numpy.array(
    [0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1,
     0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0])

doc2_vector = numpy.array(
    [1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,
     1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1])

def cosine_similarity(vector1, vector2):
    return dot(vector1, vector2) / (norm(vector1) * norm(vector2))

doc1_score = cosine_similarity(query_vector, doc1_vector)
doc2_score = cosine_similarity(query_vector, doc2_vector)
print_scores([doc1_score, doc2_score])

```

Вывод:

Relevance Scores:

```

doc1: 0.2828
doc2: 0.2828

```

Интересно... оба документа получили одинаковый рейтинг релевантности, хотя документы содержат длинные векторы с очень разным содержанием. Может быть не сразу очевидно, что происходит, поэтому давайте упростим расчет, сосредоточившись только на важных признаках.

3.1.3. Расчет сходства между разреженными векторными представлениями

Ключ к пониманию расчета в последнем разделе заключается в осознании того, что единственными релевантными признаками являются те, которые также являются общими для запроса и документа. Все остальные признаки (слова, встречающиеся в документах, которые не соответствуют запросу) не оказывают никакого влияния на то, ранжируется ли один документ выше другого. В результате мы можем удалить все остальные незначимые термины из нашего вектора, чтобы упростить пример, преобразовав плотное векторное представление в то, что известно как *разреженное векторное представление*¹, как показано на рис. 3.2.

¹ Разреженное векторное представление – это особое представление высокоразмерных векторов, в котором большинство элементов равны нулю и только несколько измерений имеют ненулевые значения. Такое представление значительно сокращает объем памяти и повышает эффективность вычислений, особенно при работе с очень высокоразмерными данными (например, при 10 000 измерений). – Прим. ред.

Разреженное
векторное
представление : [1 1]



Рис. 3.2. Разреженные векторные представления содержат только «текущую» характеристику, в отличие от плотных векторных представлений, которые также содержат записи со значением 0 для каждой характеристики

В большинстве операций по оценке поисковой системы мы, как правило, имеем дело с разреженными векторными представлениями, поскольку они более эффективны для работы при оценке на основе небольшого количества признаков.

Кроме того, мы можем еще больше упростить наши вычисления, создав векторы, включающие только «значимые записи» – термины, которые присутствуют в запросе, – как показано в следующем листинге.

Листинг 3.2. Косинусное сходство разреженных векторных представлений

```
query_vector = [1, 1] #[apple, juice]
doc1_vector  = [1, 1]
doc2_vector  = [1, 1]

doc1_score = cosine_similarity(query_vector, doc1_vector)
doc2_score = cosine_similarity(query_vector, doc2_vector)

print_scores([doc1_score, doc2_score])
```

Вывод:

```
Relevance Scores:
doc1: 1.0
doc2: 1.0
```

Обратите внимание, что doc1 и doc2 по-прежнему дают одинаковые рейтинги релевантности, но теперь оценка для каждого из них равна 1.0. Если вы помните, оценка 1.0 из расчета косинуса означает, что векторы представляют собой идеальные соответствия, и это разумно, учитывая, что оба вектора идентичны ([1, 1]).

Фактически вы заметите несколько очень интересных вещей:

- этот упрощенный расчет разреженного векторного представления по-прежнему показывает, что doc1 и doc2 возвращают эквивалентные оценки релевантности, поскольку они оба соответствуют всем словам в запросе;
- несмотря на то что абсолютная оценка между сходством плотного векторного представления (0.2828) и сходством разреженного векторного представления (1.0) различны, оценки по-прежнему одинаковы относительно друг друга в пределах каждого типа вектора;
- веса признаков для двух терминов запроса («яблоко», «сок») одинаковы для запроса и каждого из документов, что приводит к оценке косинуса 1.0.

Векторы и векторные представления

Мы были осторожны, ссылаясь на «плотные векторные представления» и «разреженные векторные представления» вместо «плотных векторов» и «разреженных векторов». Это связано с тем, что существует концептуальное различие между идеей вектора и его представления, и это различие часто вызывает путаницу.

Разреженность вектора относится к доле признаков вектора, которые имеют значимые величины. В частности, плотный вектор – это любой вектор, признаки которого имеют в основном ненулевые значения, тогда как разреженный вектор – это любой вектор, признаки которого имеют в основном нули, независимо от того, как они хранятся или представлены. С другой стороны, векторные представления имеют дело со структурами данных, используемыми для работы с векторами. Для разреженных векторов может быть расточительно выделять память и пространство для хранения всех нулей, поэтому мы часто будем использовать разреженную структуру данных (такую как инвертированный индекс) для хранения только ненулевых значений. Вот пример:

плотный вектор:

feature_1: 1.1, feature_2: 2.3, feature_3: 7.1, feature_4: 5.2, feature_5: 8.1

плотное векторное представление: [1.1, 2.3, 7.1, 5.2, 8.1]

разреженное векторное представление: N/A (вектор не разреженный, поэтому его нельзя представить разреженным)

разреженный вектор: feature_1: 1.1, feature_2: 0, feature_3: 0, feature_4: 5.2, feature_5: 0

плотное векторное представление: [1.1, 0.0, 0.0, 5.2, 0.0]

разреженное векторное представление: { 1: 1.1, 4: 5.2 } или просто [1.1, 5.2], если позиции признаков не нужны.

Поскольку разреженный вектор содержит преимущественно нули, а его соответствующее разреженное векторное представление содержит почти противоположное (только ненулевые значения), к сожалению, люди часто путают эти понятия и неправильно называют плотные векторные представления (разреженных векторов) «плотными векторами» или даже называют любой вектор со многими измерениями «плотным вектором», а с небольшим количеством измерений – «разреженным вектором». Вы можете встретить эту путаницу в другой литературе, поэтому важно знать это различие.

Поскольку все наши векторы запросов и документов являются разреженными векторами (большинство значений равны нулю, так как количество признаков равно количеству ключевых слов в поисковом индексе), имеет смысл использовать разреженное векторное представление при поиске по ключевым словам.

Поисковые системы учитывают эти проблемы, не просто рассматривая каждый признак в векторе как 1 (существует) или 0 (не существует), а вместо этого предоставляя рейтинг для каждого признака на основе того, *насколько хорошо* он ему соответствует.

3.1.4. Частота термина: измерение того, насколько хорошо документы соответствуют термину

Проблема, с которой мы столкнулись в последнем разделе, заключается в том, что признаки в наших векторах терминов показывают только то, *есть ли* в документе слово «яблоко» или «сок», а не то, насколько хорошо каждый документ представляет любой из терминов. Побочным эффектом представления каждого термина из запроса со значением 1, если он существует, является то, что и doc1, и doc2 всегда будут иметь одинаковую оценку косинусного сходства для запроса, хотя качественно doc2 является гораздо лучшим соответствием, поскольку он говорит о яблочном соке гораздо больше.

Вместо того чтобы использовать значение 1 для каждого существующего термина, мы можем эмулировать, насколько хорошо документ ему соответствует, используя *частоту термина* (TF), которая является мерой количества раз, когда термин встречается в каждом из документов. Идея здесь заключается в том, что чем чаще термин встречается в определенном документе, тем больше вероятность того, что документ больше связан с запросом.

В следующем листинге показаны векторы с подсчетом количества раз, когда каждый термин встречается в документе или запросе, в качестве весов признаков.

Листинг 3.3. Косинусное сходство необработанных векторов TF

```
query_vector    = [1, 1] #[apple:1, juice:1]
doc1_tf_vector  = [1, 1] #[apple:1, juice:1]
doc2_tf_vector  = [3, 4] #[apple:3, juice:4]

doc1_score = cosine_similarity(query_vector, doc1_tf_vector)
doc2_score = cosine_similarity(query_vector, doc2_tf_vector)

print_scores([doc1_score, doc2_score])
```

Вывод:

Relevance Scores:

```
doc1: 1.0
doc2: 0.9899
```

Вопреки тому, что вы могли бы ожидать, doc1 считается лучшим соответствием по косинусному сходству, чем doc2. Это связано с тем, что термины «яблоко» и «сок» встречаются в одинаковой пропорции (одно вхождение каждого термина на каждое вхождение другого термина) как в запросе, так и в doc1, что делает их наиболее текстуально похожими. Другими словами, хотя doc2 интуитивно больше относится к запросу, поскольку он содержит термины в запросе значительно больше, косинусное сходство возвращает doc1, так как это точное соответствие для запроса, в отличие от doc2. Поскольку наша цель со-

стоит в том, чтобы документы, такие как doc2 с более высоким TF, получали более высокие оценки, мы можем добиться этого, переключившись с косинусного сходства на другую оценочную функцию, такую как *скалярное произведение* или *евклидово расстояние*, которое увеличивается по мере увеличения весов признаков. Давайте используем скалярное произведение ($a \cdot b$), равное косинусному сходству, умноженному на длину вектора запроса и длину вектора документа: $a \cdot b = |a| \times |b| \times \cos(\theta)$. Скалярное произведение приведет к тому, что документы, содержащие больше соответствующих терминов, получают более высокую оценку, в отличие от косинусного сходства, которое дает более высокую оценку документам, содержащим более схожую долю соответствующих терминов между запросом и документами.

Соответствие фраз и другие приемы релевантности

К настоящему моменту вы, возможно, задаетесь вопросом, почему мы продолжаем рассматривать «яблоко» и «сок» как независимые термины и почему мы просто не рассматриваем «яблочный сок» как фразу, чтобы поднять документы, которые соответствуют точной фразе, выше. Соответствие фраз – один из многих простых приемов настройки релевантности, которые мы обсудим далее в этой главе. На данный момент мы сохраним простую обработку запросов и будем работать только с отдельными ключевыми словами, чтобы сосредоточиться на нашей главной цели – объяснении векторной оценки релевантности и функций текстовой оценки ключевых слов.

В следующем листинге мы заменим косинусное сходство вычислением скалярного произведения, чтобы учесть величину векторов документа (которая растет с увеличением соответствий для каждого термина запроса) при вычислении релевантности.

Листинг 3.4. Скалярное произведение векторов TF

```
query_vector    = [1, 1] #[apple:1, juice:1]
doc1_tf_vector  = [1, 1] #[apple:1, juice:1]
doc2_tf_vector  = [3, 4] #[apple:3, juice:4]

doc1_score = dot(query_vector, doc1_tf_vector)
doc2_score = dot(query_vector, doc2_tf_vector)

print_scores([doc1_score, doc2_score])
```

Вывод:

Relevance Scores:

```
doc1: 2
doc2: 7
```

Как вы можете видеть, doc2 теперь дает более высокую оценку релевантности для запроса, чем doc1, улучшение, которое лучше согла-

суется с нашей интуицией. Обратите внимание, что оценки релевантности больше не ограничены диапазоном от 0 до 1, как это было с косинусным сходством. Это связано с тем, что скалярное произведение учитывает величину векторов документа, которая может неограниченно увеличиваться с дополнительными совпадающими вхождениями¹ ключевых слов.

Хотя использование TF в качестве веса признаков в наших векторах, безусловно, помогает, текстовые запросы демонстрируют дополнительные проблемы, которые необходимо учитывать. До сих пор все наши документы содержали каждый термин из наших запросов, что не соответствует большинству реальных сценариев. Следующий пример лучше продемонстрирует некоторые ограничения, которые все еще присутствуют при использовании только взвешивания на основе частоты терминов для нашей оценки сходства разреженных векторов на основе текста. Давайте начнем со следующих трех текстовых документов:

Document 1:

In light of the big reveal in her interview, the interesting thing is that the person in the wrong probably made a good decision in the end.

Document 2:

My favorite book is the cat in the hat, which is about a crazy cat in a hat who breaks into a house and creates the craziest afternoon for two kids.

Document 3:

My careless neighbors apparently let a stray cat stay in their garage unsupervised which resulted in my favorite hat that I let them borrow being ruined.

Давайте теперь преобразуем эти документы в их соответствующие (разреженные) векторные представления и вычислим оценку сходства. Следующий листинг ранжирует сходство текста на основе необработанных TF (количества терминов).

Листинг 3.5. Ранжирование сходства текста на основе количества терминов

```
def term_count(content, term):
    tokenized_content = tokenize(content)
    term_count = tokenized_content.count(term.lower())
    return float(round(term_count, 4))

query = "the cat in the hat"
terms = tokenize(query)

query_vector = list(numpy.repeat(1, len(terms)))
doc_vectors = [[term_count(doc, term) for term in terms] for doc in
```

¹ Вхождение, англ. *occurrence*, – это появление термина в строке. Частота вхождений (TF) – это «встречаемость» термина в документе или его части. – *Прим. ред.*


```
docs]
doc_scores = [dot(v, query_vector) for v in doc_vectors]

print_term_count_scores(terms, doc_vectors, doc_scores)
```

Вывод:

```
labels: ['the', 'cat', 'in', 'the', 'hat']query vector: [1, 1, 1, 1, 1]
```

Document Vectors:

```
doc1: [5.0, 0.0, 4.0, 5.0, 0.0]
doc2: [3.0, 2.0, 2.0, 3.0, 2.0]
doc3: [0.0, 1.0, 2.0, 0.0, 1.0]
```

Relevance Scores:

```
doc1: 14.0
doc2: 12.0
doc3: 4.0
```

Хотя мы получаем разные оценки релевантности для каждого документа, основанные на количестве соответствий каждого термина, порядок результатов не обязательно соответствует нашим ожиданиям относительно того, какие документы являются лучшими соответствиями. Интуитивно мы бы ожидали следующий порядок:

- 1 *doc2*: потому что он о книге *The Cat in the Hat*,
- 2 *doc3*: потому что он соответствует всем словам «the», «cat», «in» и «hat»;
- 3 *doc1*: потому что он соответствует только словам «the» и «in», хотя он содержит их много раз.

Проблема здесь в том, что, поскольку термин считается одинаково важным каждый раз, когда он появляется, оценка релевантности без разбора увеличивается с каждым дополнительным появлением этого термина. В этом случае *doc1* получает самую высокую оценку, потому что он содержит 14 общих совпадений терминов (первое «the» пять раз, «in» четыре раза и второе «the» пять раз), что дает больше общих совпадений терминов, чем любой другой документ.

Однако не имеет смысла, что документ, содержащий эти слова 14 раз, должен считаться в 14 раз более релевантным, чем документ с одним совпадением. Вместо этого документ следует считать более релевантным, если он соответствует многим различным терминам из запроса, а не тем же терминам снова и снова. Часто реальные вычисления ТФ ослабляют эффект каждого дополнительного вхождения (occurrence) слова, вычисляя ТФ как логарифм или квадратный корень числа вхождений (встречаемости) каждого термина, как мы делаем на рис. 3.3. Кроме того, ТФ часто также нормируется относительно длины документа путем деления ТФ на общее количество терминов в каждом документе. Поскольку более длинные документы, естественно, с большей вероятностью будут содержать любой заданный термин чаще, это помогает нормировать оценку для учета этих вариаций дли-

ны документа (согласно знаменателю для рис. 3.3). Наш окончательный нормированный расчет ТФ можно увидеть на рис. 3.3.

$$TF(t, d) = \sqrt{\frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}}$$

Рис. 3.3. Нормированный расчет TF, t представляет термин, а d представляет документ. TF равен квадратному корню из числа появлений термина в текущем документе ($f_{t,d}$), деленного на число терминов в документе ($\sum_{t \in d} df_{t,d}$). Квадратный корень ослабляет дополнительный вклад релевантности каждого дополнительного появления термина, в то время как знаменатель нормирует эту ослабленную частоту по отношению к длине документа, так что более длинные документы с большим количеством терминов сопоставимы с более короткими документами с меньшим количеством терминов

Существует много вариаций расчета TF, только некоторые из которых выполняют нормировку длины документа (знаменатель) или ослабляют эффект дополнительных появлений термина (квадратный корень здесь или иногда с использованием логарифма). Например, Apache Lucene (библиотека поиска, поддерживающая Solr, OpenSearch и Elasticsearch) вычисляет TF только как квадратный корень числителя, но затем умножает его на отдельную норму длины документа (эквивалентную квадратному корню знаменателя в нашем уравнении) при выполнении определенных расчетов ранжирования.

В дальнейшем мы будем использовать этот нормированный расчет TF, чтобы гарантировать, что дополнительные вхождения того же термина продолжат повышать релевантность, но с убывающей скоростью. Следующий листинг показывает новую функцию TF в действии.

Листинг 3.6. Ранжирование схожести текста на основе TF

```
def tf(term, doc):
    tokenized_doc = tokenize(doc)
    term_count = tokenized_doc.count(term.lower())
    doc_length = len(tokenized_doc)
    return numpy.sqrt(term_count / doc_length)

query = "the cat in the hat"
terms = tokenize(query)
query_vector = list(numpy.repeat(1, len(terms)))
doc_vectors = [[tf(term, doc) for term in terms] for doc in docs]
doc_scores = [dot(dv, query_vector) for dv in doc_vectors]
print term frequency scores(terms, doc_vectors, doc_scores)
```

Вывод:

Document TF Vector Calculations:

```
doc1: [tf(doc1, "the"), tf(doc1, "cat"), tf(doc1, "in"),
        tf(doc1, "the"), tf(doc1, "hat")]
doc2: [tf(doc2, "the"), tf(doc2, "cat"), tf(doc2, "in"),
        tf(doc2, "the"), tf(doc2, "hat")]
```

```
doc3: [tf(doc3, "the"), tf(doc3, "cat"), tf(doc3, "in"),
      tf(doc3, "the"), tf(doc3, "hat")]
```

Document TF Vector Values:

```
Labels: ['the', 'cat', 'in', 'the', 'hat']
doc1: [0.4303, 0.0, 0.3849, 0.4303, 0.0]
doc2: [0.3111, 0.254, 0.254, 0.3111, 0.254]
doc3: [0.0, 0.1961, 0.2774, 0.0, 0.1961]
```

Relevance Scores:

```
doc1: 1.2456
doc2: 1.3842
doc3: 0.6696
```

Нормированная функция *tf* показывает улучшение, так как *doc2* теперь имеет самый высокий рейтинг, как и ожидалось. Это в основном из-за эффекта подавления на количество вхождений термина в *doc1* (который совпадал с «the» и «in» так много раз), так что каждое дополнительное вхождение вносит меньший вклад в вес признака, чем предыдущие вхождения. К сожалению, *doc1* по-прежнему имеет второй по величине рейтинг, так что даже улучшенного расчета TF было недостаточно, чтобы поднять более соответствующий *doc3* выше.

Следующим шагом для улучшения будет учет относительной важности терминов, так как «cat» и «hat» интуитивно важнее, чем такие распространенные слова, как «the» и «in». Давайте изменим наш расчет оценок, чтобы исправить эту оплошность, введя новую переменную, которая включает важность каждого термина.

3.1.5. Обратная частота документа: измерение важности термина в запросе

Хотя частота TF оказалась полезной для измерения того, насколько хорошо документ соответствует каждому термину в запросе, он мало что делает для дифференциации важности терминов в запросе. В этом разделе мы познакомимся с методом, использующим значимость определенных ключевых слов на основе их частоты встречаемости в документах.

Частота документа (DF) для термина определяется как общее количество документов в поисковой системе, которые содержат термин, и служит хорошим показателем важности термина. Идея здесь заключается в том, что более конкретные или редкие слова (например, «cat» и «hat»), как правило, важнее, чем общие слова (например, «the» и «in»). Функция, используемая для расчета частоты документа, показана на рис. 3.4.

$$DF(t) = \sum_{i=1}^{|D|} \text{if } t \in D_i : 1; \quad \text{if } t \notin D_i : 0$$

Рис. 3.4. Расчет частоты документа. *D* – это набор всех документов, *t* – это входной термин, а *D_i* – это *i*-й документ в *D*. Чем ниже частота документа для термина (*DF(t)*), тем более конкретным и важным является термин при его появлении в запросах

Поскольку мы хотели бы, чтобы более важные слова имели более высокий рейтинг, мы берем *обратную частоту документа* (IDF), как определено на рис. 3.5.

$$IDF(t) = 1 + \log\left(\frac{|D| + 1}{DF(t) + 1}\right)$$

Рис. 3.5. Обратная частота документа. $|D|$ – общее количество всех документов, t – термин, а $DF(t)$ – частота документа. Чем ниже $IDF(t)$, тем менее значим термин, и чем выше, тем больше термин в запросе должен учитываться для оценки релевантности

Продолжая наш пример запроса «кот в шляпе» из последнего раздела, вектор IDF будет выглядеть следующим образом.

Листинг 3.7. Вычисление обратной частоты документа

```
def idf(term):
    df_map = {"the": 9500, "cat": 100,
              "in": 9000, "hat": 50}
    total_docs = 10000
    return 1 + numpy.log((total_docs+1) / (df_map[term] + 1))

terms = ["the", "cat", "in", "the", "hat"]
idf_vector = [idf(term) for term in terms]

print_inverse_document_frequency_scores(terms, idf_vector)
```

Вывод:

IDF Vector Values:
[idf("the"), idf("cat"), idf("in"), idf("the"), idf("hat")]

IDF Vector:
[1.0513, 5.5953, 1.1053, 1.0513, 6.2786]

Подсчет количества документов, имитирующий реалистичную статистику из инвертированного индекса.

Функция IDF, которая определяет важность термина в запросе.

IDF зависит от терминов, а не от документов, поэтому он одинаков как для запросов, так и для документов.

Эти результаты выглядят обнадеживающими. Теперь термины можно взвешивать на основе их относительной описательности или значимости для запроса:

- 1 «hat»: 6.2786
- 2 . «cat»: 5.5953
- 3 3. «in»: 1.1053
- 4 4. «the»: 1.0513

Далее мы объединим методы ранжирования TF и IDF, которые вы изучили до сих пор, в сбалансированную функцию ранжирования релевантности.

3.1.6. TF-IDF: сбалансированная метрика взвешивания для текстовой релевантности

Теперь у нас есть два основных компонента текстовой релевантности:

- TF измеряет, насколько хорошо термин описывает документ;
- IDF измеряет, насколько важен каждый термин.

Большинство поисковых систем и многие другие приложения науки о данных используют комбинацию этих факторов в качестве основы для оценки текстового сходства, используя вариацию функции на рис. 3.6.

$$\text{TF-IDF} = \text{TF} \cdot \text{IDF}^2$$

Рис. 3.6. Оценка TF-IDF. Объединяет вычисления TF и IDF в сбалансированную оценку сходства текстового ранжирования

С этой улучшенной функцией взвешивания признаков мы наконец можем вычислить сбалансированную оценку релевантности следующим образом.

Листинг 3.8. Вычисление TF-IDF для запроса the cat in the hat

```
def tf_idf(term, doc):
    return TF(term, doc) * IDF(term)**2

query = "the cat in the hat"
terms = tokenize(query)
query_vector = list(numpy.repeat(1, len(terms)))
doc_vectors = [[tf_idf(doc, term) for term in terms] for doc in docs]
doc_scores = [[dot(query_vector, dv)] for dv in doc_vectors]
print_tf_idf_scores(terms, doc_vectors, doc_scores)
```

Вывод:

Document TF-IDF Vector Calculations

```
doc1: [tf_idf(doc1, "the"), tf_idf(doc1, "cat"), tf_idf(doc1, "in"),
      tf_idf(doc1, "the"), tf_idf(doc1, "hat")]
doc2: [tf_idf(doc2, "the"), tf_idf(doc2, "cat"), tf_idf(doc2, "in"),
      tf_idf(doc2, "the"), tf_idf(doc2, "hat")]
doc3: [tf_idf(doc3, "the"), tf_idf(doc3, "cat"), tf_idf(doc3, "in"),
      tf_idf(doc3, "the"), tf_idf(doc3, "hat")]
```

Document TF-IDF Vector Scores

```
Labels: ['the', 'cat', 'in', 'the', 'hat']
doc1: [0.4756, 0.0, 0.4703, 0.4755, 0.0]
doc2: [0.3438, 7.9521, 0.3103, 0.3438, 10.0129]
doc3: [0.0, 6.1399, 0.3389, 0.0, 7.7311]
```

Relevance Scores:

```
doc1: 1.4215
doc2: 18.9633
doc3: 14.2099
```

Наконец, наши результаты поиска имеют смысл – doc2 получает наивысшую оценку, так как он соответствует самым важным словам больше всего, за ним следует doc3, который содержит все слова, но не так много раз, а затем doc1, который содержит только множество незначительных слов.

Этот расчет TF-IDF лежит в основе многих алгоритмов релевантности поисковых систем, включая алгоритм сходства по умолчанию, известный как BM25, который в настоящее время используется для ранжирования на основе ключевых слов в большинстве поисковых систем. Мы познакомимся с BM25 в следующем разделе.

3.2. Управление расчетом релевантности

В последнем разделе мы показали, как запросы и документы могут быть представлены в виде векторов и как косинусное сходство или другие измерения сходства, такие как скалярное произведение, могут использоваться в качестве функции релевантности для сравнения запросов и документов. Мы представили ранжирование TF-IDF, которое можно использовать для создания веса признаков, уравнивающего как частоту встречаемости (TF), так и статистическую значимость термина (IDF) для каждого термина в векторе на основе терминов. В этом разделе мы покажем, как можно задать и контролировать полную функцию релевантности в поисковой системе, включая общие возможности запросов, моделирование запросов как функций, выбор ранжирования или фильтрации и применение различных видов бустинга.

3.2.1. BM25: стандартный алгоритм сходства текста по умолчанию

BM25 – это название алгоритма сходства по умолчанию в Apache Lucene, Apache Solr, Elasticsearch, OpenSearch и многих других поисковых системах. BM25 (сокращение от Okapi «Best Matching» версии 25) был впервые опубликован в 1994 году и демонстрирует улучшения по сравнению со стандартным ранжированием косинусного сходства TF-IDF во многих реальных текстовых оценках ранжирования. На данный момент он по-прежнему превосходит модели ранжирования, использующие эмбединги из большинства неточно настроенных LLM, поэтому он служит хорошей основой для ранжирования на основе ключевых слов.

BM25 по-прежнему использует TF-IDF в своей основе, но также включает несколько других параметров, что дает больше контроля над такими факторами, как точка насыщения TF и нормирование длины документа. Он также суммирует веса для каждого сопоставленного ключевого слова вместо вычисления косинуса.

Полный расчет BM25 показан на рис. 3.7. Переменные определены следующим образом:

- t = term; d = document; q = query;
- $freq(t, d)$ – это простой TF, $\sum_{t \in d}$ показывает, сколько раз термин t встречается в документе d ;

- $IDF(t) = \log \left(\frac{N - N(t) + 0.5}{N(t) + 0.5} + 1 \right)$,
- это вариация IDF, используемая в BM25, где N – общее количество документов, а $N(t)$ – количество документов, содержащих термин t ;
- $|d|$ – количество терминов в документе d ;
- $avgdl$ – среднее количество терминов на документ в индексе;
- k – свободный параметр, который обычно находится в диапазоне от 1.2 до 2.0 и увеличивает точку насыщения TF;
- b – свободный параметр, обычно устанавливаемый на уровне около 0.75. Он увеличивает эффект нормирования документа.

$$BM25(q, d) = \sum_{t \in q} IDF(t) \cdot \frac{freq(t, d)}{freq(t, d) + k \cdot (1 - b + b \cdot |d| / avgdl)}$$

Рис. 3.7. Оценочная функция BM25. Она по-прежнему преимущественно использует упрощенную TF и вариацию IDF, но обеспечивает больший контроль над тем, насколько каждое дополнительное вхождение термина вносит вклад в оценку (параметр k) и насколько оценки нормируются на основе длины документа (параметр b)

Вы можете видеть, что числитель содержит параметры $freq$ (упрощенная TF) и IDF , в то время как знаменатель добавляет новые параметры нормирования k и b . Точка насыщения TF контролируется k , что делает дополнительные совпадения с тем же термином меньше по мере увеличения k , и b , который контролирует уровень нормирования длины документа больше по мере его увеличения. TF для каждого термина вычисляется как $freq(t, d) / (freq(t, d) + k \cdot (1 - b + b \cdot |d| / avgdl))$, что является более сложным расчетом, чем тот, который мы использовали на рис. 3.3. Концептуально BM25 просто предоставляет эвристически определенный лучший способ нормирования TF, чем традиционный TF-IDF. Также стоит отметить, что существует несколько вариаций алгоритма BM25 (BM25F, BM25+), и в зависимости от используемой вами поисковой системы вы можете увидеть некоторые небольшие изменения и оптимизации.

Вместо того чтобы повторно реализовывать всю эту математику в Python, пока мы тестируем BM25, давайте перейдем к использованию нашей поисковой системы и посмотрим, как она выполняет расчет. Давайте начнем с создания коллекции в поисковом движке (листинг 3.9). Коллекция содержит определенную схему и конфигурацию, и это единица, которой мы будем индексировать документы, искать, ранжировать и извлекать результаты поиска. Затем мы проиндексируем некоторые документы (используя наш предыдущий пример с котом в шляпе), как показано в листинге 3.10.

Листинг 3.9. Создание коллекции `cat_in_the_hat`

```
engine = get_engine()
collection = engine.create_collection("cat_in_the_hat")
```

Вывод:

```
Wiping "cat_in_the_hat" collection
Creating "cat_in_the_hat" collection
Status: Success
```

По умолчанию установлен движок Apache Solr. Смотрите приложение В для использования других поддерживаемых поисковых систем и векторных баз данных.

Листинг 3.10. Добавление документов в коллекцию

```
docs = [{"id": "doc1",
        "title": "Worst",
        "description": "\"The interesting thing is that the person in
the
                                wrong made the right decision in the
end.\""},
        {"id": "doc2",
        "title": "Best",
        "description": "\"My favorite book is the cat in the hat, which
is
                                about a crazy cat who breaks into a house and
                                creates a crazy afternoon for two kids.\""},
        {"id": "doc3",
        "title": "Okay",
        "description": "\"My neighbors let the stray cat stay in their
that
                                garage, which resulted in my favorite hat
                                I let them borrow being ruined.\""}]
collection.add_documents(docs)
```

Вывод:

```
Adding Documents to 'cat_in_the_hat' collection
Status: Success
```

С нашими документами, добавленными в поисковую систему, мы теперь можем выполнить наш запрос и увидеть полные оценки BM25. Следующий листинг выполняет поиск по запросу `the cat in the hat` и запрашивает объяснение расчета релевантности для каждого документа.

Листинг 3.11. Ранжирование и проверка оценки сходства BM25

```
query = "the cat in the hat"
request = {"query": query,
          "query_fields": ["description"],
          "return_fields": ["id", "title", "description", "score"],
          "explain": True}
```

```
response = collection.search(**request)
display_search(query, response["docs"])
```

Вывод:

Query: the cat in the hat

Ranked Docs:

```
[{'id': 'doc2',
'title': ['Best'],
'description': ['My favorite book is the cat in the hat, which is about a
➡crazy cat who breaks into a house and creates a crazy afternoon for
➡two kids.'],
'score': 0.68231964, '[explain]':
  '0.68231964 = sum of:
    0.15655403 = weight(description:the in 1) [SchemaSimilarity], result of:
      0.15655403 = score(freq=2.0), product of:
        2.0 = boost
        0.13353139 = idf, computed as  $\log(1 + (N - n + 0.5) / (n + 0.5))$  from:
          3 = n, number of documents containing term
          3 = N, total number of documents with field
        0.58620685 = tf, computed as  $\text{freq} / (\text{freq} + k1 * (1 - b + b * dl / \text{avgdl}))$  from:
          2.0 = freq, occurrences of term within document
          1.2 = k1, term saturation parameter
          0.75 = b, length normalization parameter
          28.0 = dl, length of field
          22.666666 = avgdl, average length of field
      0.19487953 = weight(description:hat in 1) ...
      0.27551934 = weight(description:cat in 1) ...
      0.05536667 = weight(description:in in 1) ...
  }, {'id': 'doc3',
'title': ['Okay'],
'description': ['My neighbors let the stray cat stay in their garage, which
➡resulted in my favorite hat that I let them borrow being ruined.'],
'score': 0.62850046, '[explain]': '
  0.62850046 = sum of:
    0.21236044 = weight(description:the in 2) ...
    0.08311336 = weight(description:hat in 2) ...
    0.21236044 = weight(description:cat in 2) ...
    0.120666236 = weight(description:in in 2) ...
  }, {'id': 'doc1',
'title': ['Worst'],
'description': ['The interesting thing is that the person in the wrong made
➡the right decision in the end.'],
'score': 0.3132525,
'[explain]': '
  0.3132525 = sum of:
    0.089769006 = weight(description:the in 0) ...
    0.2234835 = weight(description:in in 0) ...
  }]
```

Для документа с самым высоким рейтингом, doc2, вы можете увидеть часть расчета баллов с использованием компонентов tf и idf, а также баллы высокого уровня для каждого сопоставленного термина в запросе для двух других документов. Если вы хотите углубиться в математику, можете изучить полные расчеты в блокаде Jupyter.

Хотя расчет BM25 сложнее, чем расчеты веса признаков TF-IDF, он по-прежнему использует TF-IDF в качестве основной части своего расчета. В результате рейтинг BM25 находится в том же относительном порядке, что и наши расчеты TF-IDF из листинга 3.8:

Ranked Results (Listing 3.8: TF-IDF Cosine Similarity):

```
doc2: 0.998
doc3: 0.9907
doc1: 0.0809
```

Ranked Results (Listing 3.9: BM25 Similarity):

```
doc2: 0.6878265
doc3: 0.62850046
doc1: 0.3132525
```

Наш запрос для the cat in the hat все еще можно рассматривать как вектор оценок BM25 для каждого из терминов: ["the", "cat", "in", "the", "hat"].

Что может быть неочевидно – так это то, что веса признаков для каждого из этих терминов на самом деле являются переопределяемыми функциями. Вместо того чтобы думать о нашем запросе как о наборе ключевых слов, мы можем думать о нем как о математической функции, составленной из других функций, где некоторые из этих функций принимают ключевые слова в качестве входных данных и возвращают числовые значения (оценки), которые будут использоваться при расчете релевантности. Например, наш запрос можно было бы альтернативно выразить как следующий вектор:

```
[ query("the"), query("cat"), query("in"), query("the"), query("hat") ]
```

Здесь функция запроса просто вычисляет BM25 переданного термина относительно всех оцененных документов. Таким образом, BM25 всего запроса представляет собой сумму TF-IDF каждого термина. В синтаксисе запроса Solr это будет выглядеть так:

```
{!func}query("the") {!func}query("cat") {!func}query("in")
{!func}query("the") {!func}query("hat")
```

Если мы выполним эту «функционализированную» версию запроса, то получим точно такую же оценку релевантности, как если бы мы выполнили запрос напрямую. В следующем листинге показан код, который выполняет эту версию запроса.

Листинг 3.12. Сходство текста с использованием функции запроса `query`

```
query = '{!func}query("the") {!func}query("cat") {!func}query("in")
        ➔{!func}query("the") {!func}query("hat")'
request = {"query": query,
          "query_fields": "description",
          "return_fields": ["id", "title", "score"]}
response = collection.search(**request)
display_search(query, response["docs"])
```

Вывод:

Query:

```
{!func}query("the") {!func}query("cat") {!func}query("in")
{!func}query("the") {!func}query("hat")
```

Results:

```
[{'id': 'doc2', 'title': ['Best'], 'score': 0.6823196},
 {'id': 'doc3', 'title': ['Okay'], 'score': 0.62850046},
 {'id': 'doc1', 'title': ['Worst'], 'score': 0.3132525}]
```

3.2.2. Функции, функции, везде!

Мы только что столкнулись с функцией запроса (в конце предыдущего раздела), которая выполняет расчет сходства по умолчанию (BM25) для ключевых слов. Понимание того, что каждая часть запроса на самом деле является настраиваемой оценочной функцией, открывает огромные возможности для манипулирования алгоритмом релевантности. Какие *другие* виды функций можно использовать в запросах? Можем ли мы использовать другие функции в нашем расчете оценки – возможно, некоторые, которые не основаны на тексте?

Вот частичный список функций и методов оценки, обычно применяемых для влияния на оценки релевантности:

- *геопространственный бустинг* – документы рядом с пользователем, выполняющим запрос, должны ранжироваться выше;
- *бустинг даты* – более новые документы должны получить более высокий рост релевантности;
- *бустинг популярности* – более популярные документы должны получить более высокий рост релевантности;
- *бустинг поля* – термины, соответствующие определенным полям, должны получить более высокий вес, чем в других полях;
- *бустинг категории* – документы в категориях, связанных с терминами запроса, должны получить более высокий уровень релевантности;
- *бустинг фраз* – документы, соответствующие многотерминным фразам в запросе, должны ранжироваться выше, чем те, которые соответствуют только словам по отдельности;
- *семантическое расширение* – документы, содержащие другие слова или понятия, которые тесно связаны с ключевыми словами запроса и контекстом, должны быть повышены.

Использование API поиска, не зависящего от поисковой системы

Во всей книге и кодовой базе мы реализовали набор библиотек Python, предоставляющих общий API для индексации документов (`collection.add_documents(documents)` или `collection.write(dataframe)`), запроса документов (`collection.search(**query_parameters)`) и выполнения других операций поисковой системы. Это позволяет вам выполнять один и тот же код в книге и соответствующих блокнотах независимо от того, какую поддерживаемую поисковую систему или векторную базу данных вы выбрали, делегируя создание синтаксиса, специфичного для движка, клиентской библиотеке. Смотрите приложение В для получения подробной информации о том, как легко переключаться между поисковыми системами.

Хотя эти общие методы вызова поиска на основе ИИ по сравнению с вашей любимой поисковой системой являются мощнее, в некоторых случаях также полезно увидеть детали базовой реализации в поисковой системе, а для более сложных примеров может быть даже сложно выразить всю мощь того, что происходит, используя API более высокого уровня, не зависящий от поисковой системы. По этой причине мы иногда также включаем в книгу сырой синтаксис поисковой системы для нашей поисковой системы по умолчанию (Apache Solr). Если вы не знакомы с Apache Solr и его синтаксисом, пожалуйста, не слишком увязайте в деталях. Важно понимать понятия достаточно хорошо, чтобы вы могли применять их в своей поисковой системе по вашему выбору.

Эти методы (и многие другие) поддерживаются большинством основных поисковых систем. Например, бустинг полей можно выполнить в нашем поисковом клиенте, добавив `^BOOST_AMOUNT` после любого из полей, указанных в `query_fields` для запроса:

Generic search request syntax:

```
{ "query": "the cat in the hat",
  "query_fields": ["title^10", "description^2.5"] }
```

Этот запрос обеспечивает 10-кратное повышение релевантности для совпадений в поле `title` и 2.5-кратное повышение релевантности для совпадений в поле описания. При переводе в синтаксис Solr это выглядит так:

Solr request syntax:

```
{ "query": "the cat in the hat",
  "params": { "defType": "edismax",
              "qf": "title^10 description^2.5" } }
```

Каждая поисковая система отличается от другой, но многие из этих методов встроены в специальные парсеры¹ запросов в Solr либо че-

¹ Парсинг в программировании – это автоматизированный сбор и анализ данных с помощью скриптов (парсеров). В программировании парсинг используется для перевода написанной пользователем программы в бинарный машинный код. – *Прим. ред.*

рез синтаксис запроса, либо через параметры парсера запросов, как в только что показанном парсере запросов edismax.

Бустинг по полному совпадению фраз, по фразам из двух слов и по фразам из трех слов также является встроенной функцией парсера запросов edismax Solr:

- бустинг документов, содержащих точную фразу "the cat in the hat" в поле title:

Solr request syntax:

```
{ "query": "the cat in the hat",  
  "params": { "defType": "edismax",  
              "qf": "title description",  
              "pf": "title" } }
```

- бустинг документов, содержащих фразы из двух слов "the cat", "cat in", "in the" или "the hat" в поле заголовка или описания:

Solr request syntax:

```
{ "query": "the cat in the hat",  
  "params": { "defType": "edismax",  
              "qf": "title description",  
              "pf2": "title description" } }
```

- бустинг документов, содержащих фразы из трех слов "the cat in" или "in the hat" в поле описания:

Solr request syntax:

```
{ "query": "the cat in the hat",  
  "params": { "defType": "edismax",  
              "qf": "title description",  
              "pf3": "description" } }
```

Многие другие методы бустинга релевантности требуют создания пользовательских функций с использованием запросов функций. Например, если мы хотим создать запрос, который повышает рейтинг релевантности только тех документов, которые географически ближе всего расположены к пользователю, выполняющему поиск, мы можем выполнить следующий запрос Solr:

Solr request syntax:

```
{ "query": "*",  
  "sort": "geodist(location, $user_latitude, $user_longitude) asc",  
  "params": { "user_latitude": 33.748,  
              "user_longitude": -84.39 } }
```

Последний запрос использует параметр `sort` для строгого упорядочивания документов по функции `geodist`, которая принимает имя поля локации документа вместе с широтой и долготой пользователя в качестве параметров. Это отлично работает при рассмотрении одного признака, но что, если мы хотим построить более сложную сортировку на основе многих признаков? Чтобы добиться этого, мы можем об-

новить наш запрос, чтобы применить несколько функций при вычислении оценки релевантности, а затем по этой оценке отсортировать:

Solr request syntax:

```
{ "query": "{!func}scale(query($keywords),0,25)
      ↳{!func}recip(geodist($lat_long_field,$user_latitude,
      ↳$user_longitude),1,25,1)
      ↳{!func}recip(ms(NOW/HOUR,modify_date),3.16e-11,25,1)
      ↳{!func}scale(popularity,0,25)",
  "params": { "keywords": "basketball",
              "lat_long_field": "location",
              "user_latitude": 33.748,
              "user_longitude": -84.391}}
```

Этот запрос имеет несколько интересных характеристик:

- он создает вектор запроса, содержащий четыре признака: оценку релевантности BM25 для ключевых слов (чем выше, тем лучше), географическое расстояние (чем меньше, тем лучше), дату публикации (чем новее, тем лучше) и популярность (чем выше, тем лучше);
- каждое из значений признаков масштабируется от 0 до 25, поэтому все они сопоставимы, при этом лучшая оценка для каждой характеристики составляет 25, а худшая оценка – около 0;
- таким образом, «идеальная оценка» составит 100 (все 4 характеристики оцениваются по 25), а худшая оценка будет близка или равна 0;
- поскольку относительный вклад 25 указан как часть запроса для каждого признака, мы можем легко корректировать веса для любых признаков на лету, чтобы повлиять на окончательный расчет релевантности.

С последним запросом мы полностью взяли расчет релевантности в свои руки, смоделировав признаки релевантности и присвоив им веса. Хотя это очень мощный инструмент, он все еще требует значительных ручных усилий для определения того, какие признаки наиболее важны для данной области и настройки их весов. В главе 10 мы рассмотрим создание машинно-обученных моделей ранжирования, которые автоматически принимают эти решения для нас (процесс, известный как *обучение ранжированию*). На данный момент наша цель – просто понять механику моделирования признаков в векторах запросов и то, как программно управлять их весами.

Более глубокое погружение в запросы функций

Если вы хотите узнать больше о том, как использовать запросы функций Solr, мы рекомендуем прочитать главу 7 книги «Solr в действии», одной из наших предыдущих книг, написанной Треем Грейнджером и Тимоти Поттером (Manning, 2014; <https://mng.bz/n0Y5>). Полный список доступных запросов функций в Solr вы также можете посмотреть в документации в разделе запросов функций в справочном руководстве Solr (<https://mng>.

[bz/vlop](#)). Если вы используете другую поисковую систему, ознакомьтесь с ее документацией для получения аналогичных рекомендаций.

Мы увидели силу использования функций в качестве признаков в наших запросах, но до сих пор все наши примеры были так называемыми аддитивными бустингами, где сумма значений каждого вычисления функции составляет окончательную оценку релевантности. Также часто бывает полезно объединять функции менее четким, более гибким способом с помощью мультипликативного бустинга, который мы рассмотрим в следующем разделе.

3.2.3. Выбор мультипликативного или аддитивного бустинга для функций релевантности

Последняя тема для рассмотрения, касающаяся того, как мы управляем нашими функциями релевантности, – это выбор между мультипликативным или аддитивным бустингом признаков релевантности.

Во всех наших примерах до этого момента мы добавляли несколько признаков в наш вектор запроса для внесения вклада в оценку. Например, все следующие запросы Solr дадут эквивалентные вычисления релевантности, предполагая, что они отфильтрованы до одного и того же набора результатов (т. е. `filters=["the cat in the hat"]`):

Text query (score + filter):

```
{"query": "the cat in the hat"}
```

Function Query (score only, no filter):

```
{"query": ' {!func}query("the cat in the hat")' }
```

Multiple Function Queries (score only, no filter):

```
{"query": ' {!func}query("the")
           ↳{!func}query("cat")
           ↳{!func}query("in")
           ↳{!func}query("the")
           ↳{!func}query("hat")' }
```

Boost Query (score only, no filter):

```
{"query": "*",
  "params": {"bq": "the cat in the hat"}}
```

Вид бустинга релевантности в каждом из этих примеров известен как *аддитивный бустинг*, и это хорошо соответствует нашему понятию запроса как не более чем вектора признаков, которые должны иметь сходство при сравнении документов. При аддитивном бустинге относительный вклад каждого признака уменьшается по мере добавления большего количества признаков, поскольку общая оценка – это просто сумма оценок всех признаков.

Мультипликативный бустинг, напротив, позволяет масштабировать (умножать) всю вычисленную оценку релевантности документа на одну или несколько функций. Мультипликативный бустинг позволяет усилениям (бустам) «наслаиваться» друг на друга, предотвращая необ-

ходимость индивидуального ограничения весов различных разделов запроса, как мы делали в разделе 3.2.2. Там нам нужно было гарантировать, что оценки ключевых слов, географическое расстояние, возраст и популярность документов были масштабированы до 25 % от оценки релевантности, чтобы они в сумме давали максимальную оценку 100 %.

Чтобы применить мультипликативный бустинг в Apache Solr, вы можете использовать парсер запросов `boost` (синтаксис: `{!boost ...}`) в векторе запроса или, если вы используете парсер запросов `edismax`, упрощенный параметр запроса `boost`. Следующие два запроса умножат оценку релевантности документа на значение в поле `popularity`, умноженное на десять:

```
{"query": "the cat in the hat",  
  "params": {"defType": "edismax",  
             "boost": "mul(popularity,10)"}}
```

```
{"query": "{!boost b=mul(popularity,10)} the cat in the hat"}
```

В этом примере запрос для `the cat in the hat` по-прежнему использует аддитивный бустинг (значение `BM25` каждого ключевого слова суммируется), но окончательная оценка значения в поле `popularity` умножается на 10. Этот мультипликативный бустинг позволяет популярности масштабировать оценку релевантности независимо от любых других признаков.

В целом мультипликативный бустинг предлагает вам большую гибкость для комбинирования различных функций релевантности без необходимости явно предопределять формулу релевантности, учитывающую каждый потенциальный способствующий фактор. С другой стороны, эта гибкость может привести к неожиданным последствиям, если значения мультипликативного бустинга для определенных признаков станут слишком высокими и затмят другие признаки. Напротив, аддитивные бусты могут быть сложными в управлении, поскольку вам нужно явно масштабировать их так, чтобы их можно было объединить, сохраняя при этом предсказуемый вклад в общую оценку. Однако с этим явным масштабированием вы сохраняете жесткий контроль над расчетом оценки релевантности и диапазоном оценок. Как аддитивный, так и мультипликативный бустинг могут быть полезны, поэтому лучше рассмотреть имеющуюся задачу и поэкспериментировать с тем, что даст вам наилучшие результаты. Мы рассмотрели основные способы управления рейтингом релевантности в поисковой системе, но сопоставление и фильтрация документов часто могут быть столь же важны, поэтому мы рассмотрим их в следующем разделе.

3.2.4. Различение сопоставления (фильтрации) и ранжирования (оценки) документов

Мы говорили о запросах и документах как о векторах признаков, но до сих пор мы в основном обсуждали поиск как процесс либо вычисления сходства векторов (косинуса или скалярного произведения), либо

сложения оценок документов для каждого признака (ключевого слова или функции) в запросе.

После индексации документов выполнение запроса включает два основных шага:

- *сопоставление* (matching) – фильтрацию результатов по известному набору возможных ответов;
- *ранжирование* (ranking) – упорядочивание всех возможных ответов по релевантности.

Часто мы можем полностью пропустить первый шаг (сопоставление) и по-прежнему видеть те же самые результаты на первой странице (и для многих страниц), поскольку наиболее релевантные результаты должны, как правило, иметь самый высокий рейтинг и, таким образом, отображаться первыми. Если вспомнить главу 2, мы даже видели некоторые вычисления векторной оценки (сравнение векторов признаков для продуктов питания, например «яблочный сок» и «пончик»), где мы вообще не смогли бы фильтровать результаты. Вместо этого нам пришлось сначала оценить каждый документ, чтобы определить, какие из них возвращать, основываясь только на релевантности. В этом сценарии (с использованием плотных векторных эмбедингов) у нас даже не было ключевых слов или других атрибутов, которые можно было бы использовать в качестве фильтра.

Итак, если начальная фаза сопоставления фактически необязательна, зачем ее вообще делать? Один очевидный ответ заключается в том, что она обеспечивает значительную оптимизацию производительности. Вместо того чтобы перебирать каждый отдельный документ и вычислять оценку релевантности, мы можем значительно ускорить как наши вычисления релевантности, так и общее время отклика нашей поисковой системы, сначала отфильтровав начальные результаты по меньшему набору документов, которые являются логическими соответствиями.

Есть дополнительные преимущества возможности фильтровать наборы результатов, поскольку мы можем предоставлять аналитику, такую как количество соответствующих друг другу документов или количество определенных значений, найденных в документах (известных как *грани* или *агрегации*). Возвращение фасетов и аналогичных агрегированных метаданных из результатов поиска помогает пользователю впоследствии фильтровать по определенным значениям для дальнейшего изучения и уточнения своего набора результатов. Наконец, существует множество сценариев, в которых «наличие логических соответствий» следует рассматривать как одну из наиболее важных признаков в функции ранжирования, поэтому простая фильтрация по логическим соответствиям заранее может значительно упростить расчет релевантности. Мы обсудим эти компромиссы в следующем разделе.

3.2.5. Логическое соответствие: взвешивание взаимосвязей между терминами в запросе

Мы только что упомянули, что фильтрация результатов перед их оценкой – это в первую очередь оптимизация производительности и что первые несколько страниц результатов поиска, скорее всего, будут выглядеть одинаково независимо от того, фильтруете ли вы результаты или просто выполняете ранжирование по релевантности.

Однако это справедливо только в том случае, если ваша функция релевантности успешно содержит признаки, которые уже соответствующим образом повышают лучшие логические соответствия. Например, рассмотрим разницу между ожиданиями для следующих запросов:

- 1 "statue of liberty"
- 2 statue AND of AND liberty
- 3 statue OR of OR liberty
- 4 statue of liberty

С точки зрения логического соответствия первый запрос будет очень точным, сопоставляя только документы, содержащие *точную* фразу «statue of liberty». Второй запрос будет сопоставлять только документы, содержащие все термины «statue», «of» и «liberty», но не обязательно как фразу. Третий запрос будет сопоставлять любой документ, содержащий любой из трех терминов, что означает, что документы, содержащие *только* «of», будут совпадать, но документы, содержащие «statue» и «liberty», должны ранжироваться намного выше из-за расчета баллов BM25.

Теоретически, если бустинг фраз включен как функция, документы, содержащие полную фразу, скорее всего, будут ранжироваться выше всего, за ними следуют документы, содержащие все термины, а затем документы, содержащие любое из слов. Если предположить, что это произойдет, вы должны увидеть похожий порядок результатов независимо от того, фильтруете ли вы их по логическим булевым совпадениям или сортируете только на основе функции релевантности.

На практике пользователи часто считают, что логическая структура их запросов в высокой степени соответствует документам, которые они ожидают увидеть, поэтому соблюдение этой логической структуры и фильтрация *перед* ранжированием позволяют вам удалять результаты, которые, как указывают запросы пользователей, можно безопасно удалить.

Однако иногда логическая структура пользовательских запросов неоднозначна, как в нашем четвертом примере: запрос statue of liberty. Означает ли это логически statue AND of AND liberty, statue OR of OR liberty или что-то более тонкое, например (statue AND of) OR (statue AND liberty) OR (of AND liberty), что, по сути, означает «соответствие как минимум двум из трех терминов». Использование параметра «минимальное соответствие» (min_match) в нашем API поиска позволяет вам легко контролировать эти типы пороговых значений соответствия, даже на основе запроса:

- 100 % терминов запроса должны совпадать (эквивалентно `statue AND of AND liberty`):

Generic search request syntax:

```
{ "query": "statue of liberty",
  "min_match": "100%" }
```

Solr request syntax:

```
{ "query": "statue of liberty",
  "params": { "defType": "edismax",
              "mm": "100%" } }
```

- по крайней мере один термин запроса должен совпадать (эквивалентно `statue OR of OR liberty`):

Generic search request syntax:

```
{ "query": "statue of liberty",
  "min_match": "1" }
```

Solr request syntax:

```
{ "query": "statue of liberty",
  "params": { "defType": "edismax",
              "mm": "1" } }
```

- по крайней мере два термина запроса должны совпадать (эквивалентно `(statue AND of) OR (statue AND liberty) OR (of AND liberty)`):

Generic search request syntax:

```
{ "query": "statue of liberty", "query_parser": "edismax",
  "min_match": "2" }
```

Solr request syntax:

```
{ "query": "statue of liberty",
  "params": { "defType": "edismax",
              "mm": "2" } }
```

Параметр `min_match` в нашем API Python поддерживает указание либо минимального процента (от 0 до 100 %) терминов, либо количества терминов (от 1 до N терминов), которые должны совпадать. Этот параметр соответствует параметру `mm` Solr и параметру `minimum_should_match` OpenSearch и Elasticsearch. Помимо принятия процента или количества терминов для сопоставления, эти системы также поддерживают ступенчатую функцию, например `mm=2<-30%5<3`. Этот пример функции `step`¹ означает, что «все термины обязательны, если терминов меньше 2, до 30 % терминов могут отсутствовать, если терминов меньше 5, и по крайней мере 3 термина должны существовать, если терминов 5

¹ Step function в программировании – это функция, которая представляет последовательность операций, происходящих линейным образом. Она не управляет состояниями, а выполняет серию шагов, которые могут не зависеть от текущего состояния системы. – *Прим. ред.*

или более». При использовании Solr параметр `mm` работает с парсером запросов `edismax`, который является основным парсером запросов, который мы будем использовать для запросов на сопоставление текста в этой книге, если Solr настроен как ваш движок (согласно приложению В). Вы можете ознакомиться с разделом «Расширенные параметры DisMax» Справочного руководства Solr для получения более подробной информации о том, как точно настроить ваши логические правила сопоставления с этими минимальными возможностями сопоставления (<https://mng.bz/mRo8>).

При размышлении о построении функций релевантности идеи фильтрации и оценки часто могут путаться, особенно потому, что большинство поисковых систем выполняют и то и другое для своего основного параметра запроса. Мы попытаемся разделить эти задачи в следующем разделе.

3.2.6. Разделение задач: фильтрация и оценка

В разделе 3.2.4 мы провели различие между идеями сопоставления и ранжирования. Сопоставление результатов является логическим и реализуется путем фильтрации результатов поиска по подмножеству документов, тогда как ранжирование результатов является качественным и реализуется путем оценки всех документов относительно запроса и последующей сортировки их по этой вычисленной оценке. В этом разделе мы рассмотрим некоторые методы, обеспечивающие максимальную гибкость в управлении сопоставлением и ранжированием путем четкого разделения задач фильтрации и оценки.

Наш поисковый API имеет два основных способа управления фильтрацией и оценкой: параметры `query` и `filters`. Рассмотрим следующий запрос:

Generic search request syntax:

```
{
  "query": "the cat in the hat",
  "query_fields": ["description"],
  "filters": [{"category": "books"}, {"audience": "kid"}],
  "min_match": "100%"
}
```

Solr request syntax:

```
{
  "query": "the cat in the hat",
  "filters": ["category:books", "audience:kid"],
  "params": {
    "qf": ["description"],
    "mm": "100%",
    "defType": "edismax"
  }
}
```

В этом запросе поисковой системе предписывается отфильтровать возможный набор результатов, оставив только документы со значением «books» в поле `category` и значением «kid» в поле `audience`. Однако в дополнение к этим фильтрам запрос также действует как фильтр, поэтому набор результатов дополнительно фильтруется, оставляя только документы, содержащие (100 %) значений «the», «cat», «in» и «hat» в поле `description`.

Логическое различие между параметрами `query` и `filters` заключается в том, что `filters` действуют только как фильтр, тогда как `query`

действует как *фильтр и вектор признаков* для ранжирования релевантности. Такое двойное использование параметра запроса является полезным поведением по умолчанию для запросов, но смешивание задач фильтрации и оценки в одном параметре может быть неоптимальным для более сложных запросов, особенно если мы просто пытаемся манипулировать расчетом релевантности, а не произвольно удалять результаты из нашего набора документов.

Есть несколько способов решения этой задачи:

- смоделируйте параметр запроса как функцию (функции учитываются только для релевантности и не фильтруют):

Solr request syntax:

```
{
  "query": '{!func}query("{!edismax qf=description mm=100%
    v=$user_query}")',
  "filters": "{!cache=false v=$user_query}",
  "params": {"user_query": "the cat in the hat"}}

```

- сделайте так, чтобы ваш запрос соответствовал всем документам (без фильтрации или оценки) и примените параметр запроса `boost (bq)` для влияния на релевантность без оценки:

Solr request syntax:

```
{
  "query": "*",
  "filters": "{!cache=false v=$user_query}",
  "params": {"bq": "{!edismax qf=description mm=100% v=$user_query}",
    "user_query": "the cat in the hat"}}

```

Параметр `query` и фильтрует, и затем усиливает на основе релевантности, `filters` фильтрует только фильтры, а `bq` только усиливает. Два предыдущих подхода логически эквивалентны, но мы рекомендуем второй вариант, так как немного чище использовать выделенный параметр `bq`, который был разработан для внесения вклада в расчет релевантности без фильтрации.

Вы могли заметить, что обе версии запроса также содержат запрос фильтра `{!cache=false v=$user_query}`, который фильтрует по `user_query`. Поскольку параметр `query` намеренно больше не фильтрует наши результаты поиска, теперь требуется параметр `filters`, если мы все еще хотим фильтровать по введенному пользователем запросу. Специальный параметр `cache=false` используется для отключения кеширования фильтра. Кеширование фильтров в Solr включено по умолчанию, поскольку фильтры, как правило, часто повторно используются в разных запросах. Поскольку параметр `user_query` вводится пользователем и в этом случае сильно варьируется (нечасто повторно используется в разных запросах), нет смысла засорять кеш поисковой системы этими значениями. Если вы попытаетесь отфильтровать введенные пользователем запросы, не отключив кеш, это приведет к трате системных ресурсов и, скорее всего, замедлит работу вашей поисковой системы.

Главная тема здесь заключается в том, что можно четко разделить логическую фильтрацию от функций ранжирования, чтобы сохранить полный контроль и гибкость в контроле над результатами поиска. Хотя

выполнение этих усилий может быть излишним для простого текстового ранжирования, разделение этих задач становится критически важным при попытке построить более сложные функции ранжирования.

Теперь, когда вы понимаете механику того, как построить эти виды специально созданных функций ранжирования, давайте завершим эту главу кратким обсуждением того, как применять эти методы для реализации ранжирования пользовательской и домен-специфичной релевантности.

3.3. Реализация пользовательского и домен-специфичного ранжирования релевантности

В разделе 3.2 мы рассмотрели, как динамически изменять параметры нашего алгоритма подобия запроса к документу. Это включало передачу наших собственных функций в качестве признаков, которые вносят вклад в оценку, в дополнение к текстовому ранжированию релевантности.

Хотя текстовое ранжирование релевантности с использованием BM25, TF-IDF, векторного косинусного сходства или какого-либо другого подхода на основе статистики к вхождению слов может обеспечить приличную общую релевантность поиска «из коробки», оно не может противостоять хорошим факторам домен-специфичной релевантности. Вот некоторые домен-специфичные факторы, которые часто имеют наибольшее значение в различных доменах:

- поиск ресторанов – географическая близость, диетические ограничения пользователя, вкусовые предпочтения пользователя, ценовой диапазон;
- поиск новостей – свежесть (дата), популярность, географическая область;
- электронная коммерция – вероятность конверсии (переход по ссылке, добавление в корзину и/или покупка);
- поиск фильмов – совпадение имени (название, актер и т. д.), популярность фильма, дата выхода, оценка критиков;
- поиск работы – должность, уровень должности, диапазон компенсации, географическая близость, отрасль работы;
- поиск в интернете – совпадение ключевых слов на странице, популярность страницы, популярность веб-сайта, расположение совпадения на странице (в заголовке, заголовке, тексте и т. д.), качество страницы (дублирующий контент, спам-контент и т. д.), соответствие темы между страницей и запросом.

Это всего лишь примеры, но большинство поисковых систем и доменов обладают уникальными особенностями, которые необходимо учитывать для обеспечения оптимального опыта поиска. Эта глава едва затронула поверхность бесчисленных способов, которыми вы можете управлять функциями сопоставления и ранжирования, чтобы возвращать лучший контент. Существует целая профессия – называемая инже-

нерией релевантности, – которая во многих организациях посвящена настройке релевантности поиска. Если вы хотите углубиться, мы настоятельно рекомендуем одну из наших предыдущих книг, «Релевантный поиск» Дуга Тернбулла и Джона Берримана (Manning, 2016), которая является руководством по этому виду инженерии релевантности.

Каждый поисковый движок и домен имеют уникальные особенности, которые необходимо учитывать для предоставления оптимального опыта поиска. Вместо того чтобы вручную моделировать эти особенности релевантности, поисковый движок на базе ИИ может использовать машинное обучение для автоматического создания и взвешивания таких особенностей.

Цель этой главы – дать вам знания и инструменты, которые понадобятся в следующих главах для влияния на рейтинг релевантности, поскольку мы начинаем интегрировать больше автоматизированных методов машинного обучения. Мы начнем применять это в нашей следующей главе о краудсорсинговой релевантности.

Резюме

- Мы можем представлять запросы и документы как плотные или разреженные числовые векторы и назначать документам ранг релевантности на основе расчета сходства векторов (например, косинусного сходства).
- Использование TF-IDF или расчетов сходства BM25 (также основанных на TF-IDF) для наших оценок сходства текста обеспечивает более значимую меру важности признаков (ключевых слов) в наших запросах и документах, позволяя улучшить ранжирование текста по сравнению с простым просмотром совпадений терминов.
- Оценка сходства текста – это одна из многих функций, которые мы можем вызывать как признак в наших запросах для ранжирования релевантности. Мы можем вводить функции в наши запросы вместе с сопоставлением ключевых слов и оценкой, поскольку каждая ключевая фраза фактически является просто функцией ранжирования.
- Рассмотрение фильтрации и оценки как отдельных задач обеспечивает лучший контроль при указании наших собственных функций ранжирования.
- Для оптимизации релевантности нам необходимо как создавать домен-специфичные функции релевантности, так и использовать специфичные для пользователя признаки, вместо того чтобы полагаться только на сопоставление ключевых слов и ранжирование.

4

Краудсорсинговая релевантность

В этой главе рассматривается:

- использование коллективных предпочтений ваших пользователей для повышения релевантности вашей поисковой платформы;
- сбор и работа с поведенческими сигналами пользователей;
- использование отраженного интеллекта для создания самонастраивающихся моделей;
- создание сквозной модели бустинга сигналов.

В главе 1 мы представили такие измерения намерения пользователя, как понимание контента, понимание пользователя и понимание предметной области (домена). Чтобы создать оптимальную поисковую платформу на базе ИИ, нам нужно уметь объединять каждый из этих контекстов, чтобы понимать намерение запроса наших пользователей. Однако вопрос в том, как мы получаем это понимание?

Мы можем учиться из многих источников информации: документов, баз данных, внутренних графов знаний, поведения пользователей, экспертов предметной области и т. д. В некоторых организациях есть команды, которые вручную помечают документы метками тем или категорий, а некоторые даже передают эти задачи на аутсорсинг с помощью таких инструментов, как Amazon Mechanical Turk, что по-

звоняет им получать ответы от людей по всему миру. Для выявления вредоносного поведения или ошибок на веб-сайтах компании часто позволяют своим пользователям сообщать о проблемах и даже предлагать исправления. Все это примеры краудсорсинга – опоры на вклад многих людей для получения новой информации.

Когда дело доходит до релевантности поиска, краудсорсинг может играть жизненно важную роль, хотя обычно важно не раздражать ваших ценных клиентов, постоянно прося их о помощи. К счастью, часто можно неявно учиться у ваших пользователей, основываясь на их поведении. Например, чтобы обнаружить наиболее релевантные документы для запроса, мы можем изучить журналы, чтобы определить документы, на которые чаще всего кликали другие пользователи при запуске того же поиска. Эти клики дают сигналы о том, какие результаты наиболее релевантны для запроса.

В этой главе мы рассмотрим, как мы можем собирать, анализировать и генерировать идеи из этих сигналов для краудсорсинга релевантности. Мы также рассмотрим процесс отраженного интеллекта, представив три ключевых типа моделей для популяризированной релевантности (бустинг сигналов), для персонализированной релевантности (совместная фильтрация) и для обобщенной релевантности (обучение ранжированию). Вы также проиндексируете набор данных электронной коммерции и создадите свою первую модель отраженного интеллекта.

4.1. Работа с пользовательскими сигналами

Каждый раз, когда клиент совершает действие, например отправляет запрос или покупает продукт, это дает сигнал о намерении этого пользователя. Мы можем регистрировать и обрабатывать эти сигналы, чтобы узнать больше о каждом пользователе, различных группах пользователей или всей нашей пользовательской базе.

В этом разделе представлена мощь использования пользовательских сигналов с образцом набора данных электронной коммерции, который мы будем использовать на протяжении всей книги, и он пройдет вас через механизмы сбора, хранения и обработки этих сигналов.

4.1.1. Контент, сигналы и модели

При создании поисковых систем на релевантность поиска влияют два источника данных высокого уровня: контент и сигналы. Большая часть контента представлена в форме документов, которые могут представлять собой веб-страницы, списки продуктов, компьютерные файлы, изображения, видео, факты или любой другой тип информации, доступной для поиска. Документы с контентом обычно содержат текстовые или встроенные поля, которые используются для поиска, а также другие поля, представляющие атрибуты, связанные с контентом (автор, размер, цвет, даты и т. д.). Определяющей характеристикой документов с контентом является то, что они содержат информацию, которую пользователи ищут, и в идеале ответы на их запросы.

Когда пользователь видит контент в ответ на запрос, он может кликнуть на результат, добавить его в корзину или выполнить некоторые другие действия. Эти действия являются сигналами, и они играют ключевую роль в предоставлении информации о том, как пользователи взаимодействуют с контентом. Эти сигналы впоследствии можно объединить и использовать для построения моделей, чтобы улучшить релевантность ваших алгоритмов сопоставления и ранжирования. Определяющей характеристикой сигналов является то, что они представляют собой предоставленную пользователем информацию для демонстрации того, как пользователи хотят взаимодействовать с контентом.

Иногда также может быть полезно полагаться на внешние источники данных – или *модели* – как часть опыта поиска. Это может включать запрос графа знаний, ссылку на список сущностей или вызов большой языковой модели (LLM) или другой базовой модели, обученной на внешних источниках данных. Эти внешние модели могут использоваться для лучшей интерпретации пользовательских запросов, обоснования и понимания контента или даже для резюмирования или генерации нового контента для возврата. Хотя мы можем рассматривать модели как третий источник данных для нашей поисковой системы, они обучаются на контенте и/или сигналах и, таким образом, служат производным и уточненным представлением этих двух исходных источников данных.

Подводя итог, можно сказать, что мы используем три основных источника информации для улучшения поиска: атрибуты предметов (контент), наблюдаемые взаимодействия пользователя с контентом (сигналы) и внешние модели (которые выводятся из контента и/или сигналов).

Для многих задач, которые мы решаем при построении поиска на основе ИИ, мы можем получить схожие результаты, используя либо контент, либо сигналы, но они дают нам две разные перспективы релевантности. В идеальных случаях мы можем применять обе точки зрения для создания еще более умной системы, но полезно понимать их сильные и слабые стороны, чтобы наилучшим образом их использовать.

Например, пытаясь найти синоним для слова «driver», мы можем просмотреть текстовое содержимое на предмет слов, которые часто встречаются в одних и тех же документах. В этом случае мы можем найти слова (в порядке приоритета по проценту документов, в которых они появляются), такие как «taxi» (40 %), «car» (35 %), «golf» (15 %), «club» (12 %), «printer» (3 %), «linux» (3 %) и «windows» (1 %). Аналогичным образом мы можем посмотреть на сигналы от пользователей, которые искали «driver», и объединить общие ключевые слова из их других поисков в порядке приоритета, такие как «screwdriver» (50 %), «printer» (30 %), «windows» (25 %), «mac» (15 %), «golf» (2 %) и «club» (2 %). Списки, полученные из сигналов и контента, могут быть похожими, но могут выглядеть совсем по-разному. Подход, основанный на контенте, сообщает нам наиболее представленные значения в наших документах, тогда как подход, основанный на сигналах, сообщает нам наиболее представленные значения, которые ищут наши пользователи.

Поскольку наша конечная цель – предоставить пользователям то, что они ищут, часто эффективнее полагаться на значения, полученные из сигналов, чем на значения, полученные из контента. Но что, если у нас нет хорошего контента, который соответствует значению, полученному из сигналов? Используем ли мы значение, полученное из контента, или пытаемся предложить другие связанные поиски на основе данных сигналов? Что, если у нас недостаточно сигналов или данные сигналов не очень чистые? Можем ли мы как-то очистить данные, полученные из сигналов, используя данные, полученные из контента?

Мы сталкиваемся с похожими проблемами, когда имеем дело с рекомендациями. Рекомендации, основанные на контенте, используют атрибуты в документах, но не понимают пользователей, тогда как рекомендации, основанные на сигналах, не понимают атрибуты контента и не будут работать без достаточного взаимодействия. Рекомендации, основанные на контенте, могут основываться на признаках, которые не важны для пользователей, тогда как рекомендации, основанные на сигналах, могут создавать самоусиливающиеся циклы, в которых пользователи взаимодействуют только с теми предметами, что им рекомендуются, но эти предметы рекомендуются только потому, что пользователи взаимодействуют с ними.

В идеале мы хотим создать сбалансированную систему, которая может использовать лучшее из интеллекта, полученного из контента и сигналов. Хотя эта глава в основном посвящена интеллекту, полученному из сигналов, краудсорсинговому интеллекту, главная цель этой книги – показать вам, как сбалансировать и объединить оба подхода для получения более оптимального опыта поиска на основе ИИ.

4.1.2. Настройка наших наборов данных о продуктах и сигналах (RetroTech)

Мы будем использовать различные наборы данных в этой книге, поскольку мы исследуем различные варианты использования, но также важно иметь наглядный пример, на который мы можем опираться по мере нашего продвижения в работе. Мы выиграем, имея надежный вариант использования поиска с большим количеством данных и взаимодействий с пользователем, и мы настроим это в разделе далее.

Стоит отметить, что большинство методов, представленных в этой книге, применимы практически ко всем вариантам поиска. Решающий фактор того, когда использовать тот или иной метод, обычно больше зависит от объема и разнообразия контента и сигналов, чем от конкретного варианта использования.

Поиск в электронной коммерции предоставляет один из наиболее конкретных вариантов использования ценности методов поиска на основе ИИ, и это также одна из наиболее хорошо понятных задач среди потенциальных читателей, поэтому мы создали набор данных электронной коммерции, чтобы помочь нам исследовать эту область: набор данных RetroTech.

Пример использования RetroTech

В условиях агрессивной конкуренции среди розничных продавцов, продающих передовую электронику, мультимедиа и технологические продукты, малому онлайн-бизнесу трудно конкурировать. Однако нишевый, но развивающийся сегмент населения предпочитает избегать новейших и лучших продуктов и вместо этого возвращается к знакомым технологиям прошлых десятилетий. Компания RetroTech была основана, чтобы удовлетворить потребности этой уникальной группы потребителей, предлагая винтажное оборудование, программное обеспечение и мультимедийные продукты, которые может быть трудно найти на современных полках.

Давайте загрузим набор данных для компании RetroTech, чтобы мы могли начать изучать связи между документами и пользовательскими сигналами, а также то, как краудсорсинговая разведка может улучшить релевантность нашего поиска.

Загрузка каталога продуктов

На сайте RetroTech представлено около 50 000 продуктов, доступных для продажи, которые нам нужно загрузить в нашу поисковую систему. Если вы создали кодовую базу AI-Powered Search для запуска примеров из главы 3, то ваша поисковая система уже запущена и работает. В противном случае инструкции по созданию и запуску всех примеров книги можно найти в приложении А.

После того как ваша поисковая система запущена, следующее, что вам нужно сделать, – это загрузить набор данных RetroTech, который сопровождает эту книгу. Набор данных включает два файла CSV, один из которых содержит все продукты RetroTech, а другой – данные сигналов за один год от пользователей RetroTech. Следующий листинг показывает несколько строк набора данных каталога продуктов, чтобы вы могли ознакомиться с форматом.

Листинг 4.1. Изучение каталога продуктов RetroTech

```
"upc","name","manufacturer","short_description","long_description"
"096009010836","Fists of Bruce Lee - Dolby - DVD", , ,
"043396061965","The Professional - Widescreen Uncut - DVD", , ,
"085391862024","Pokemon the Movie: 2000 - DVD", , ,
"067003016025","Summerbreeze - CD","Nettwerk", ,
"731454813822","Back for the First Time [PA] - CD","Def Jam South", ,
"024543008200","Big Momma's House - Widescreen - DVD", , ,
"031398751823","Kids - DVD", , ,
"037628413929","20 Grandes Exitos - CD","Sony Discos Inc.", ,
"060768972223","Power Of Trinity (Box) - CD","Sanctuary Records", ,
```

Вы можете видеть, что продукты идентифицируются по UPC (универсальному коду продукта), а также имеют название, производителя и краткое описание (используется как тизер в результатах поиска) и длинное описание (полное описание, используемое на страницах сведений о продукте).

Поскольку мы пытаемся искать продукты, наш следующий шаг – отправить их в поисковую систему для индексации. Чтобы включить поиск в нашем каталоге продуктов RetroTech, давайте запустим код индексации документов в следующем листинге, чтобы отправить документы о продукте в поисковую систему.

Листинг 4.2. Отправка документов продукта в поисковую систему

```
products_collection = engine.create_collection("products")
products_dataframe = load_dataframe("data/retrotech/products.csv")
products_collection.write(products_dataframe)
```

Вывод:

```
Wiping "products" collection
Creating "products" collection
Status: Success
Loading Products
Schema:
Root
|-- upc: long (nullable = true)
|-- name: string (nullable = true)
|-- manufacturer: string (nullable = true)
|-- short_description: string (nullable = true)
|-- long_description: string (nullable = true)
Successfully written 48194 documents
```

Наконец, чтобы убедиться, что документы теперь индексируются и доступны для поиска, давайте запустим пример поиска по ключевым словам. Следующий листинг показывает пример поиска для `ipod`, настоящего классического устройства!

Листинг 4.3. Выполнение поиска по каталогу продукции

```
def product_search_request(query, param_overrides={}):
    request = {"query": query,
               "query_fields": ["name", "manufacturer", "long_
description"],
               "return_fields": ["upc", "name", "manufacturer",
                                "short_description", "score"],
               "limit": 5,
               "order_by": [("score", "desc"), ("upc", "asc")]}
    return request | param_overrides
query = "ipod"
request = product_search_request(query)
response = products_collection.search(**request)
display_product_search(query, response["docs"])
```

Результаты предыдущего поиска `ipod` показаны на рис. 4.1, они демонстрируют, что наши продукты теперь индексируются и доступны для поиска. К сожалению, релевантность результатов довольно низкая.



Name: Fusion - Apple® iPod® Dock for Most Fusion 600 Series Stereos
Manufacturer: Fusion



Name: Alesis - Multimix 8-Channel USB 2.0 Mixer
Manufacturer: Alesis



Name: Apple® - iPod touch® 8GB* MP3 Player (4th Generation - Latest Model) - Black
Manufacturer: Apple®



Name: Apple - USB Charge/Sync Cable for Apple® iPod® shuffle
Manufacturer: Apple



Name: Yamaha - Apple® iPod® and iPhone® Dock for Most Yamaha A/V Receivers
Manufacturer: Yamaha

Рис. 4.1. Результаты поиска продуктов. Мы видим, что каталог продуктов был проиндексирован, и запрос для `ipod` теперь возвращает результаты поиска

Хотя качество ранжирования результатов поиска пока не очень хорошее, у нас есть готовая поисковая система «соответствия ключевым словам», которую мы можем начать улучшать. Мы будем использовать ее в качестве основы и начнем внедрять более интеллектуальные функции поиска на основе ИИ в оставшейся части книги. Наш следующий шаг – представить наши данные сигналов.

Загрузка данных сигналов

Поскольку RetroTech работает на вашем компьютере, ни один живой пользователь не ищет, не нажимает или иным образом не генерирует сигналы. Вместо этого мы сгенерировали набор данных, приблизительно соответствующий виду активности сигналов, который вы ожидаете в аналогичных реальных наборах данных.

Для простоты мы сохраним наши сигналы в поисковой системе, чтобы к ним можно было получить доступ как в сценариях поиска в реальном времени, так и для внешней обработки. Запуск следующего листинга смоделирует и проиндексирует некоторые образцы сигналов, которые мы можем использовать в оставшейся части главы.

Листинг 4.4. Индексация набора данных пользовательских сигналов

```
signals_collection = engine.create_collection("signals")
signals_collection.write(from_csv("data/retrotech/signals.csv"))
```


Загрузив все данные о продукте RetroTech и сигналах, мы вскоре начнем изучать способы использования данных сигналов для повышения релевантности поиска. Давайте сначала познакомимся с данными сигналов, чтобы понять, как сигналы структурируются, используются и собираются в реальных системах.

4.1.3. Изучение данных сигналов

Различные типы сигналов имеют разные атрибуты, которые необходимо записывать. Для сигнала «запрос» мы хотим записать ключевые слова пользователя. Для сигнала «клик» мы хотим записать, какой документ был кликнут, а также какой запрос привел к клику. Для последующего анализа мы также хотим записать, какие документы были возвращены и, возможно, просмотрены пользователем после запроса.

Чтобы сделать примеры более расширяемыми и избежать пользовательского кода для каждого нового типа сигнала, в этой книге мы приняли общий формат для представления сигналов. Этот формат может отличаться от того, как вы в настоящее время регистрируете сигналы, но пока вы можете в конечном итоге сопоставить свои сигналы с этим форматом, весь код в этой книге должен работать без необходимости внесения изменений, специфичных для конкретного варианта использования.

Формат сигналов, который мы используем в этой книге, следующий:

- `query_id` – уникальный идентификатор для сигнала запроса, который создал этот сигнал;
- `user` – идентификатор, представляющий конкретного пользователя поисковой системы;
- `type` – какой тип сигнала («запрос», «клик», «покупка» и т. д.);
- `target` – контент, к которому применяется сигнал в это `signal_time`;
- `signal_time` – дата и время появления сигнала.

В качестве примера предположим, что пользователь выполнил следующую последовательность действий:

- 1 выполнил запрос для `ipad` и получил три документа (`doc1`, `doc2` и `doc3`);
- 2 кликнул на `doc1`;
- 3 вернулся и кликнул на `doc3`;
- 4 добавил `doc3` в корзину;
- 5 вернулся и выполнил поиск для `ipad cover` и получил два документа (`doc4` и `doc5`);
- 6 кликнул на `doc4`;
- 7 добавил `doc4` в корзину;
- 8 купил продукты в корзине (`doc3` и `doc4`). Эти взаимодействия приведут к сигналам, показанным в табл. 4.1.

Таблица 4.1. Пример формата сигналов

query_id	user	type	target	signal_time
1	u123	query	ipad	2024-05-01-09:00:00
1	u123	results	doc1,doc2,doc3	2024-05-01-09:00:00
1	u123	click	doc1	2024-05-01-09:00:10
1	u123	click	doc3	2024-05-01-09:00:29
1	u123	add-to-cart	doc3	2024-05-01-09:03:40
2	u123	query	ipad cover	2024-05-01-09:04:00
2	u123	results	doc4,doc5	2024-05-01-09:04:00
2	u123	click	doc4	2024-05-01-09:04:40
2	u123	add-to-cart	doc4	2024-05-01-09:05:50
1	u123	purchase	doc3	2024-05-01-09:07:15
2	u123	purchase	doc4	2024-05-01-09:07:15

Следует отметить несколько моментов относительно формата сигналов.

- Тип «query» и тип «results» разбиты на отдельные сигналы. Это не обязательно, так как они происходят одновременно, но это позволяет нам сохранять структуру таблицы единообразной и не добавлять дополнительный столбец результатов, который будет применяться только к сигналу запроса. Кроме того, если пользователь нажимает ссылку «Следующая страница» или прокручивает страницу вниз и видит дополнительные результаты, эта структура позволяет нам создать новый сигнал без необходимости возвращаться и изменять исходный сигнал.
- Каждый сигнал привязывается к query_id исходного сигнала «query», который начал серию взаимодействий с контентом. Идентификатор запроса query_id – это не просто ссылка на ключевые слова, введенные пользователем, а ссылка на конкретный сигнал «query», идентифицирующий экземпляр ключевых слов запроса пользователя с меткой времени. Поскольку результаты для одних и тех же ключевых слов запроса могут со временем меняться, это позволяет нам выполнять более сложную обработку того, как пользователи реагировали на определенный набор результатов, которые им были показаны в ответ на запрос.
- Большинство типов сигналов содержат один предмет в target, но тип сигнала «results» содержит упорядоченный список документов. Порядок результатов будет иметь значение для некоторых алгоритмов, которые мы представим позже в книге, для измерения релевантности. Поэтому важно сохранить точный порядок результатов поиска. Целью в этом случае является упорядоченный список документов, а не один документ.

- Оформление заказа (checkout) привело к отдельному сигналу «purchase» (покупка) для каждого предмета вместо одного сигнала «checkout». Это делается для того, чтобы мы могли отслеживать, была ли каждая покупка произведена из отдельных запросов. Тип сигнала «checkout» можно было бы дополнительно добавить для отслеживания транзакции и, возможно, указать две покупки в качестве цели, но это излишне для наших нужд в этой книге.

Имея эти необработанные сигналы в качестве наших строительных блоков, мы теперь можем начать думать о том, как мы могли бы связать эти сигналы вместе, чтобы начать изучать наших пользователей и их интересы. В следующем разделе мы обсудим способы моделирования пользователей, сеансов и запросов в нашей поисковой платформе.

4.1.4. Моделирование пользователей, сеансов и запросов

В последнем разделе мы рассмотрели структуру пользовательских сигналов как список независимых взаимодействий, привязанных к исходному запросу. Мы предположили, что «user» присутствует с уникальным ID, но как идентифицировать и отслеживать уникального пользователя? Кроме того, как только вы определили, как отслеживать уникальных пользователей, как лучше всего разбить их взаимодействия на сеансы, чтобы понять, когда их контекст мог измениться?

Концепция пользователя в веб-поиске может быть довольно изменчивой. Если в вашей поисковой системе есть аутентифицированные (вошедшие в систему) пользователи, то у вас уже есть внутренний идентификатор пользователя для их отслеживания. Однако, если ваш поисковый движок поддерживает неаутентифицированный доступ или является общедоступным, у вас будет много пользователей, выполняющих поиск без формального идентификатора пользователя. Это не значит, что вы не можете отслеживать их, однако; это просто требует более гибкой интерпретации того, что означает «user». Единый ID позволяет нам связывать несколько сигналов от одного и того же пользователя, чтобы узнать их модели взаимодействия. Если мы представим отслеживаемую информацию как иерархию от самого прочного представления пользователя к наименее прочному, то она будет выглядеть примерно так:

- User ID – уникальный идентификатор пользователя, который сохраняется на всех устройствах (аутентифицированный);
- ID устройства – уникальный идентификатор, который сохраняется на всех сеансах на одном устройстве (например, идентификатор устройства или IP-адрес плюс отпечаток пальца устройства);
- ID браузера – уникальный идентификатор, который сохраняется на всех сеансах только в одном приложении или браузере (постоянный ID cookie);
- ID сеанса – уникальный идентификатор, который сохраняется на протяжении одного сеанса (например, cookie в режиме инкогнито браузера);
- ID запроса – уникальный идентификатор, который сохраняется только для одного запроса (браузер с отключенными cookie).

В большинстве современных поисковых приложений и, конечно, в большинстве приложений электронной коммерции нам обычно приходится иметь дело со всеми из них. Как правило, вы хотите привязать пользователя к более прочному идентификатору – тому, который находится как можно выше в списке. Связи между ID запросов и ID сеансов, а также связи между ID сеансов и ID браузеров осуществляются через cookie-файл пользователя, поэтому в конечном итоге ID браузера (постоянный уникальный ID, хранящийся в cookie-файле) является общим знаменателем для каждого из них.

В частности,

- если у пользователя включены постоянные файлы cookie, один ID браузера может иметь много ID сеансов, которые могут иметь много ID запросов;
- если пользователь очищает файлы cookie после каждого сеанса (например, используя режим инкогнито), каждый ID браузера имеет только один ID сеанса, который может иметь много ID запросов;
- если пользователь отключает файлы cookie, то каждый ID запроса имеет новый ID сеанса и новый ID браузера.

При создании поисковых платформ большинство организаций не планируют и не проектируют должным образом свои механизмы отслеживания сигналов. Если они не могут сопоставить запросы посетителей с последующими действиями, становится сложно максимально использовать возможности своей поисковой платформы на базе ИИ. В некоторых случаях можно получить недостающую информацию об отслеживании сигналов постфактум (например, путем моделирования сигналов в вероятные сеансы с использованием временных меток), но обычно лучше всего заранее спроектировать систему, чтобы лучше обрабатывать отслеживание пользователей и предотвратить потенциальную потерю информации. В следующем разделе мы обсудим, как эти богатые сигналы можно использовать для повышения релевантности с помощью процесса, известного как отраженный интеллект.

4.2. Знакомство с отраженным интеллектом

В последнем разделе мы рассмотрели, как улавливать сигналы от пользователей, когда они взаимодействуют с нашей поисковой системой. Хотя сами сигналы полезны для понимания того, как используется наша поисковая система, они также служат основными входными данными для построения моделей, которые могут постоянно учиться на взаимодействиях пользователей и позволяют нашей поисковой системе самостоятельно настраивать свою модель релевантности. В этом разделе мы расскажем, как работают эти самонастраивающиеся модели с помощью концепции отраженного интеллекта.

4.2.1. Что такое отраженный интеллект?

Представьте, что вы сотрудник хозяйственного магазина. Кто-то спрашивает вас, где можно найти молоток, и вы отвечаете ему: «Во втором ряду». Через несколько минут вы видите, как тот же человек идет из второго ряда в пятый без молотка, а затем выходит из пятого ряда с молотком в руках. На следующий день кто-то еще просит молоток, вы снова говорите: «Проход два» и наблюдаете почти идентичную модель поведения. Вы были бы плохим сотрудником, если бы не заметили эту модель и не скорректировали свои рекомендации в будущем, чтобы обеспечить лучший опыт для ваших клиентов.

К сожалению, большинство поисковых систем работают таким образом по умолчанию – у них есть в основном статический набор документов, которые возвращаются на каждый запрос независимо от того, кто каждый пользователь или как предыдущие пользователи отреагировали на показанный список документов. Однако, применяя машинное обучение к собранным сигналам, мы можем узнать о намерениях пользователей и отразить эти знания для улучшения будущих результатов поиска. Этот процесс называется *отраженным интеллектом*.

Отраженный интеллект – это создание циклов обратной связи, которые постоянно обучаются и совершенствуются на основе развивающихся взаимодействий пользователей. Рисунок 4.2 демонстрирует общий обзор процесса отраженного интеллекта.

Процесс отраженного интеллекта

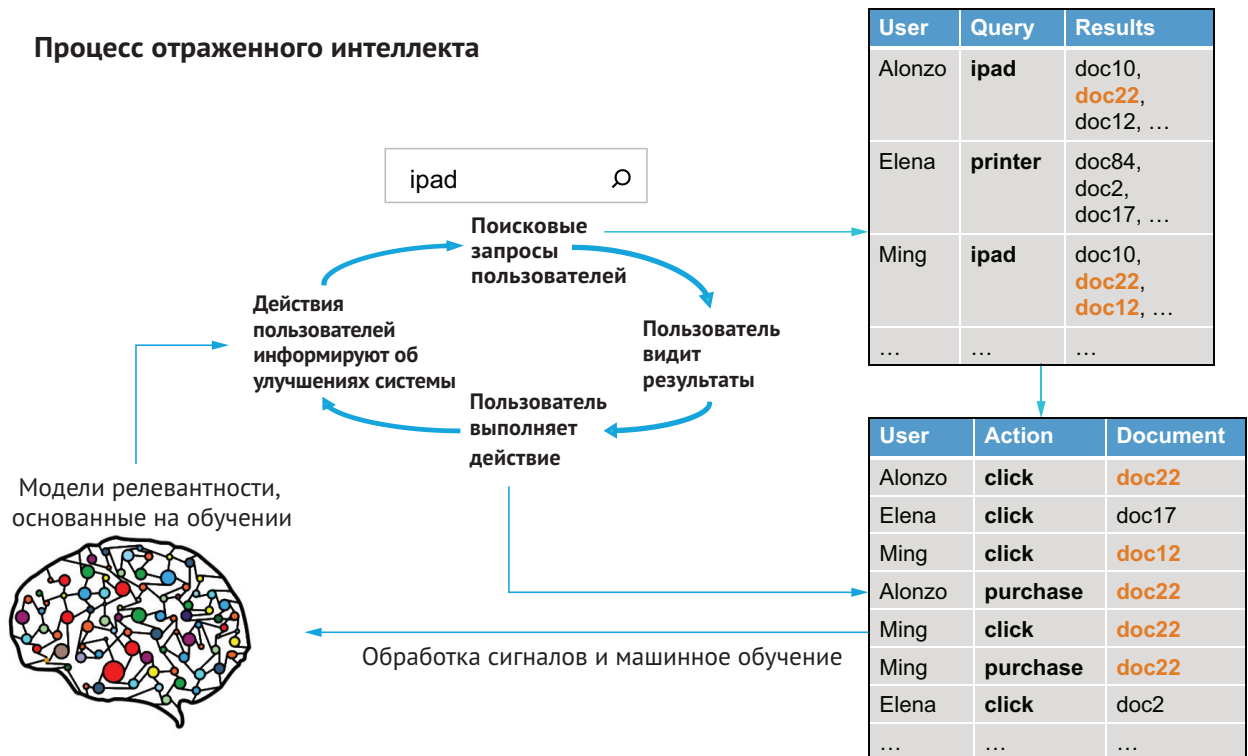


Рис. 4.2. Процесс отраженного интеллекта. Пользователь отправляет запрос, видит результаты и выполняет ряд действий. Затем эти действия (сигналы) обрабатываются для создания изученных моделей релевантности, которые улучшают будущие поиски

На рис. 4.2. пользователь (Алонзо) запускает поиск, вводя запрос `ipad` в поле поиска. Регистрируется сигнал запроса, содержащий список всех результатов поиска, отображаемых для Алонзо. Затем Алонзо видит список результатов поиска и выполняет два действия: кликает на документ (`doc22`), а затем покупает продукт, который представляет документ. Эти два дополнительных действия регистрируются как дополнительные сигналы. Все сигналы Алонзо, а также сигналы от каждого другого пользователя затем могут быть объединены и обработаны различными алгоритмами машинного обучения для создания изученных моделей релевантности.

Эти изученные модели релевантности могут повышать самые популярные результаты для определенных запросов, персонализировать результаты для каждого пользователя или даже узнавать, какие атрибуты документа, как правило, наиболее важны для всех пользователей. Модели также могут научиться лучше интерпретировать запросы пользователей, например выявлять распространенные орфографические ошибки, фразы, синонимы или другие лингвистические шаблоны и домен-специфичную терминологию.

После того как эти изученные модели релевантности будут сгенерированы, их можно будет развернуть обратно в поисковую систему и немедленно применить для улучшения результатов будущих запросов. Затем процесс начинается снова, когда следующий пользователь запускает поиск, видит (теперь, надеюсь, улучшенные) результаты поиска и взаимодействует с этими результатами. Этот процесс создает самообучающуюся систему, которая улучшается с каждым дополнительным взаимодействием пользователя, становясь со временем все умнее и релевантнее и автоматически подстраиваясь по мере развития интересов и контента пользователя.

В следующих разделах мы рассмотрим несколько категорий моделей отраженного интеллекта, включая бустинг сигналов (популяризованную релевантность), совместную фильтрацию (персонализированную релевантность) и обучение ранжированию (обобщенную релевантность). Начнем с одной из самых простых и эффективных моделей: модели бустинга сигналов.

4.2.2. Популяризованная релевантность посредством бустинга сигналов

Самые популярные запросы, отправленные в вашу поисковую систему, как правило, также являются наиболее важными для оптимизации с точки зрения релевантности. К счастью, поскольку более популярные запросы генерируют больше сигналов, мы часто можем объединять и повышать релевантность документов с наибольшим количеством сигналов для каждого запроса. Этот вид *популяризованной релевантности* (popularized relevance) известен как *бустинг сигналов* (signals boosting). Это одна из самых простых форм отраженного интеллекта, а также одна из самых эффективных для повышения релевантности ваших самых популярных, самых объемных запросов. Следующий ли-

стинг демонстрирует готовый поиск с запросом **ipad** в нашей поисковой системе RetroTech до применения бустинга сигналов.

Листинг 4.5. Выполнение поиска по ключевому слову для продуктов, соответствующих **ipad**

```
query = "ipad"
request = product_search_request(query)
response = products_collection.search(**request)display_product_
search(query, response["docs"])
```

Как и ожидалось, этот запрос возвращает много документов, содержащих ключевое слово **ipad**, причем документы, содержащие **ipad**, обычно ранжируются выше всего. На рис. 4.3 показаны результаты для этого запроса.

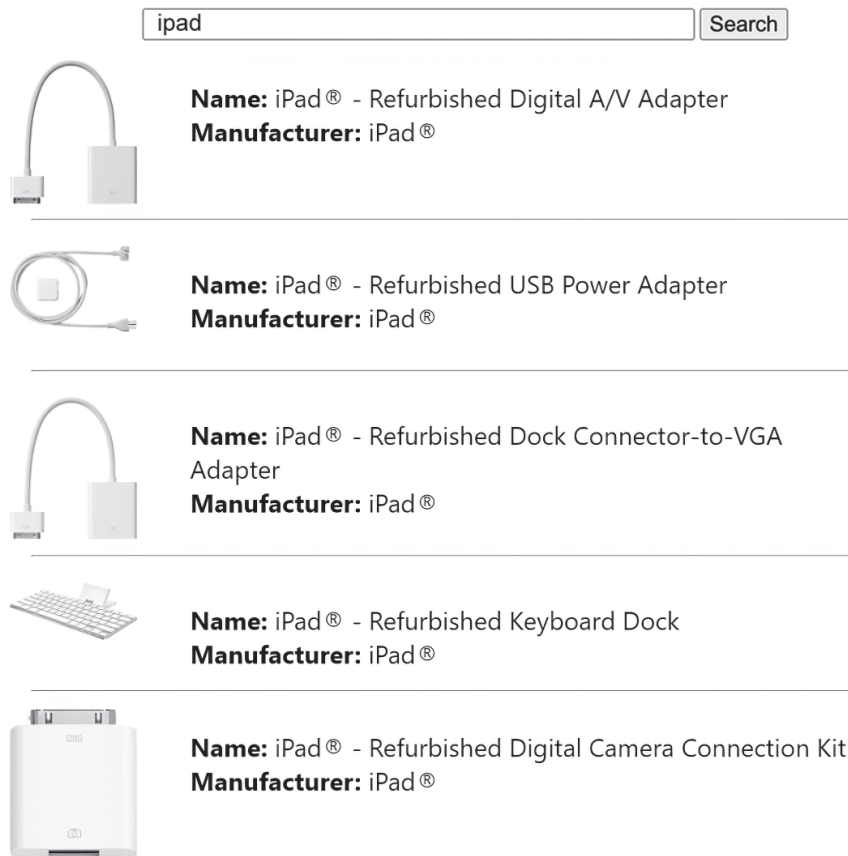


Рис. 4.3. Результаты поиска по ключевым словам на запрос **ipad**. Результаты возвращаются в первую очередь на основе количества вхождений ключевого слова, поэтому аксессуары, упоминающие ключевое слово несколько раз, ранжируются выше, чем фактический продукт, который пользователь намеревался увидеть

Хотя все эти результаты содержат слово «**ipad**» в своем содержании несколько раз, большинство пользователей будут разочарованы этими результатами, поскольку они являются вторичными аксессуарами, а не основным типом продукта, который был в центре внимания поиска. Это существенное ограничение при ранжировании только по тексту документа. Однако для очень популярных запросов многие кли-

енты, вероятно, будут многократно запускать одни и те же запросы и продираться через разочаровывающие результаты поиска, чтобы найти фактические продукты, которые они ищут. Мы можем связать эти повторные поиски в цикл обратной связи, который постоянно обновляет модель бустинга сигналов на основе новых сигналов, как показано на рис. 4.4.

Цикл обратной связи с бустингом сигналов

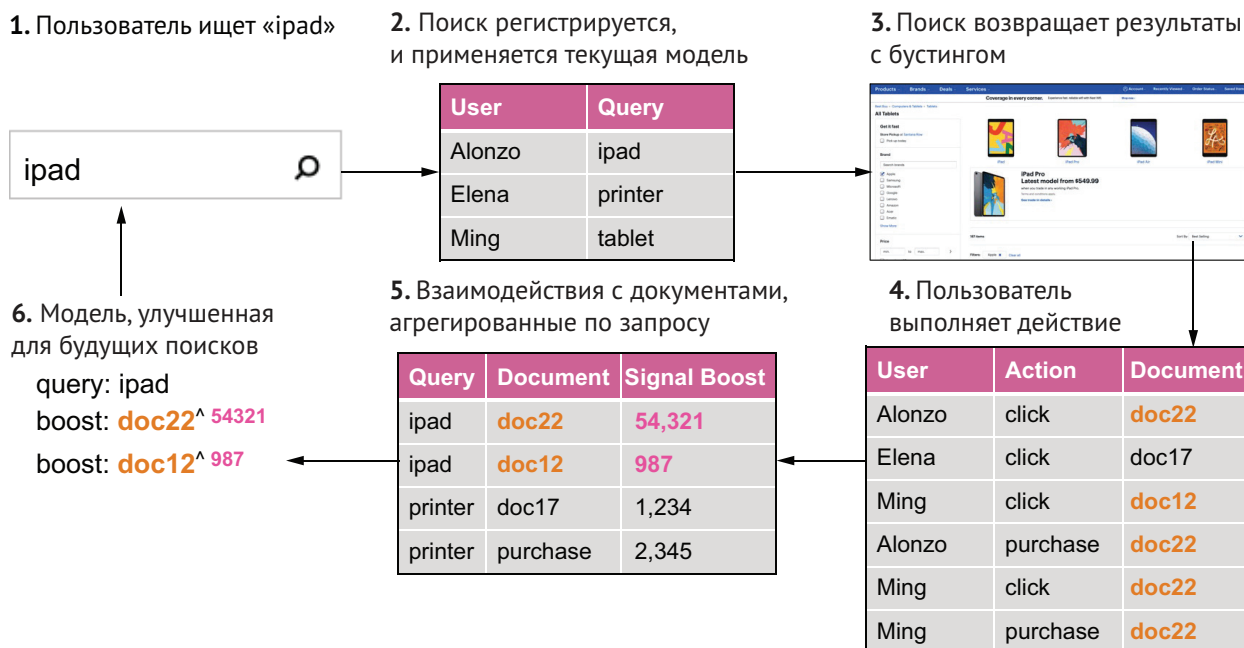


Рис. 4.4. Цикл обратной связи с бустингом сигналов. Поиск пользователя регистрируется, и текущая модель бустинга сигналов применяется для возврата результатов с бустингом. После того как пользователи выполняют действия с этими результатами, все сигналы взаимодействия пользователя с документами агрегируются исходным запросом для создания обновленной модели для дальнейшего улучшения будущих поисков

Коллекции sidecar

Коллекции sidecar¹ – это дополнительные коллекции, которые находятся в вашей поисковой системе рядом с основной коллекцией и содержат другие полезные данные для улучшения вашего поискового приложения. В нашем примере электронной коммерции наша основная коллекция – это продукты. Наша коллекция signals, которая уже была добавлена, может считаться коллекцией sidecar. Мы добавим еще одну коллекцию sidecar, signals_boosting, которую будем использовать во время запроса для улучшения наших запросов. На протяжении всей книги мы представим много других коллекций sidecar для хранения входных данных и выходных данных наших сгенерированных моделей.

¹ Sidecar в программировании – это шаблон проектирования в программной архитектуре, применяемый, в частности, для создания и развертывания микросервисов. Суть его в том, что рядом с основным контейнером приложения развертывается дополнительная служба. Этим вспомогательным контейнером базовая функциональность приложения расширяется, но не изменяется. – Прим. ред.

Листинг 4.6. Создание модели бустинга сигналов путем агрегации сигналов

```

signals_collection = engine.get_collection("signals")
create_view_from_collection(signals_collection, "signals")

signals_aggregation_query = """

SELECT q.target AS query, c.target AS doc,
COUNT(c.target) AS boost
FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
WHERE c.type = 'click' AND q.type = 'query'
GROUP BY q.target, doc
ORDER BY boost DESC"""

dataframe = spark.sql(signals_aggregation_query)
signals_boosting_collection = \
    engine.create_collection("signals_boosting")
signals_boosting_collection.write(dataframe)

```

Создает представление, чтобы сделать коллекцию сигналов доступной для запросов с помощью SQL.

Подсчитывает общее количество кликов для каждого документа по каждому ключевому слову.

Выполняет SQL-запрос агрегации сигналов.

Записывает результаты в новую коллекцию signals_boosting.

Самая важная часть листинга 4.6 – `signals_aggregation_query`, определяемая как SQL-запрос для удобства чтения. Для каждого запроса мы получим список документов, которые пользователи кликали в результатах поиска по этому запросу, а также количество кликов на документ. Упорядочивая документы по количеству кликов для каждого запроса, мы получаем упорядоченный список документов, ранжированных по популярности.

Идея здесь заключается в том, что пользователи склонны выбирать продукты, которые, по их мнению, наиболее релевантны, поэтому, если бы мы сделали бустинг этих документов, мы бы ожидали, что наши главные результаты поиска станут более релевантными. Мы проверим эту теорию в следующем листинге, используя эти агрегированные подсчеты в качестве сигналов повышения. Давайте вернемся к нашему предыдущему запросу для iPad.

Листинг 4.7. Поиск с бустингом сигналов для повышения релевантности

```

def search_for_boosts(query, collection, query_field="query"):
    boosts_request = {"query": query,
                      "query_fields": [query_field],
                      "return_fields": ["query", "doc", "boost"],
                      "limit": 10,
                      "order_by": [("boost", "desc")]}
    response = collection.search(**boosts_request)
    return response["docs"]

def create_boosts_query(boost_documents):
    print(f"Boost Documents: \n{boost_documents}")

```

```

boosts = " ".join([f'"{b["doc"]}"^{b["boost"]}'
                    for b in boost_documents])
print(f"\nBoost Query: \n{boosts}\n")
return boosts

query = "ipad"
boost_docs = search_for_boosts(query, signals_boosting_collection)
boosts_query = create_boosts_query(boost_docs)
request = product_search_request(query)
request["query_boosts"] = boosts_query

response = products_collection.search(**request)
display_product_search(query, response["docs"])

```

Усиленные документы:

```

[{"query": "ipad", "doc": "885909457588", "boost": 966},
 {"query": "ipad", "doc": "885909457595", "boost": 205},
 {"query": "ipad", "doc": "885909471812", "boost": 202},
 {"query": "ipad", "doc": "886111287055", "boost": 109},
 {"query": "ipad", "doc": "843404073153", "boost": 73},
 {"query": "ipad", "doc": "635753493559", "boost": 62},
 {"query": "ipad", "doc": "885909457601", "boost": 62},
 {"query": "ipad", "doc": "885909472376", "boost": 61},
 {"query": "ipad", "doc": "610839379408", "boost": 29},
 {"query": "ipad", "doc": "884962753071", "boost": 28}]

```

Усиленные запросы:

```

"885909457588"^966 "885909457595"^205 "885909471812"^202
"886111287055"^109"843404073153"^73 "635753493559"^62 "885909457601"^62
"885909472376"^61 "610839379408"^29 "884962753071"^28

```

Запрос в листинге 4.7 делает две примечательные вещи:

- он запрашивает коллекцию `sidecar signals_boosting` для ранжированных документов по `boost` и преобразует эти `signals boosts` в другой запрос;
- затем он передает этот бустинг-запрос в поисковую систему как `boost` во время запроса в параметре `query_boosts`. В случае Solr (наш поисковый движок по умолчанию) это внутренне преобразуется в параметр `boost` в `sum(1,query($boost_query))`, добавляемый к поисковому запросу для умножения оценки релевантности на 1 (чтобы она всегда увеличивалась) плюс рассчитанная оценка релевантности `boost_query` (см. раздел 3.2, если вы хотите освежить в памяти информацию о влиянии на ранжирование таким образом с помощью функций и мультипликативных бустов).

Если вы помните из рис. 4.3, наш исходный поиск по ключевым словам для `ipad` в основном возвращал аксессуары для iPad, а не реальные устройства iPad. Рисунок 4.5 демонстрирует улучшенные результаты после применения бустинга сигналов к этому исходному запросу.



Name: iPad® - Refurbished Digital A/V Adapter
Manufacturer: iPad®



Name: iPad® - Refurbished USB Power Adapter
Manufacturer: iPad®



Name: iPad® - Refurbished Dock Connector-to-VGA Adapter
Manufacturer: iPad®



Name: iPad® - Refurbished Keyboard Dock
Manufacturer: iPad®



Name: iPad® - Refurbished Digital Camera Connection Kit
Manufacturer: iPad®

Рис. 4.5. Результаты поиска с включенным бустингом сигналов. Вместо аксессуаров для iPad, которые отображались раньше, теперь мы видим настоящие iPad, поскольку мы собрали ответы на основе документов, с которыми пользователи решили взаимодействовать

Новые результаты значительно лучше, чем результаты только по ключевым словам. Теперь мы видим продукты, которые пользователь, скорее всего, искал, – iPad! Вы можете ожидать увидеть похожие улучшения для большинства других популярных запросов в вашей поисковой системе. Конечно, по мере продвижения вниз по списку популярных продуктов улучшения релевантности от бустинга сигналов начнут снижаться, а при недостаточном количестве сигналов мы можем даже снизить релевантность. К счастью, мы представим много других методов для повышения релевантности ответов на запросы с недостаточным объемом сигналов.

Целью этого раздела было провести вас через начальный конкретный пример реализации сквозной модели отраженного интеллекта. Агрегация сигналов, использованная в этой реализации, была очень простой, хотя результаты говорят сами за себя. Существует множество соображений и нюансов, которые следует учитывать при внедрении модели бустинга сигналов: нужно ли усиливать во время запроса или во время индексирования, как увеличить вес новых сигналов по сравнению со старыми сигналами, как избежать попыток злонамеренных пользователей повысить определенные продукты в результатах поиска путем генерации поддельных сигналов, как вводить и смешивать сигналы из разных источников и т. д. Мы подробно рассмотрим каждую из этих тем в главе 8.

Давайте перейдем от популярной релевантности и бустинга сигналов к другим типам моделей отраженного интеллекта.

4.2.3. Персонализированная релевантность через совместную фильтрацию

Давайте теперь рассмотрим метод отраженного интеллекта, называемый совместной фильтрацией (collaborative filtering, коллаборативной фильтрацией), который мы назовем *персонализированной релевантностью*. В то время как популярная релевантность определяет, какие результаты обычно наиболее популярны среди многих пользователей, персонализированная релевантность фокусируется на определении того, какие предметы, скорее всего, будут релевантны для конкретного пользователя.

Совместная фильтрация – это процесс использования наблюдений о предпочтениях некоторых пользователей для прогнозирования предпочтений других пользователей. Это самый популярный тип алгоритма, используемый рекомендательными системами, и он является источником обычных списков рекомендаций «пользователям, которым понравился этот предмет, также понравились следующие предметы», которые появляются на многих веб-сайтах. Рисунок 4.6 демонстрирует, как совместная фильтрация следует этому же циклу обратной связи отраженного интеллекта, который мы видели для моделей бустинга сигналов.

Совместная фильтрация (рекомендации)

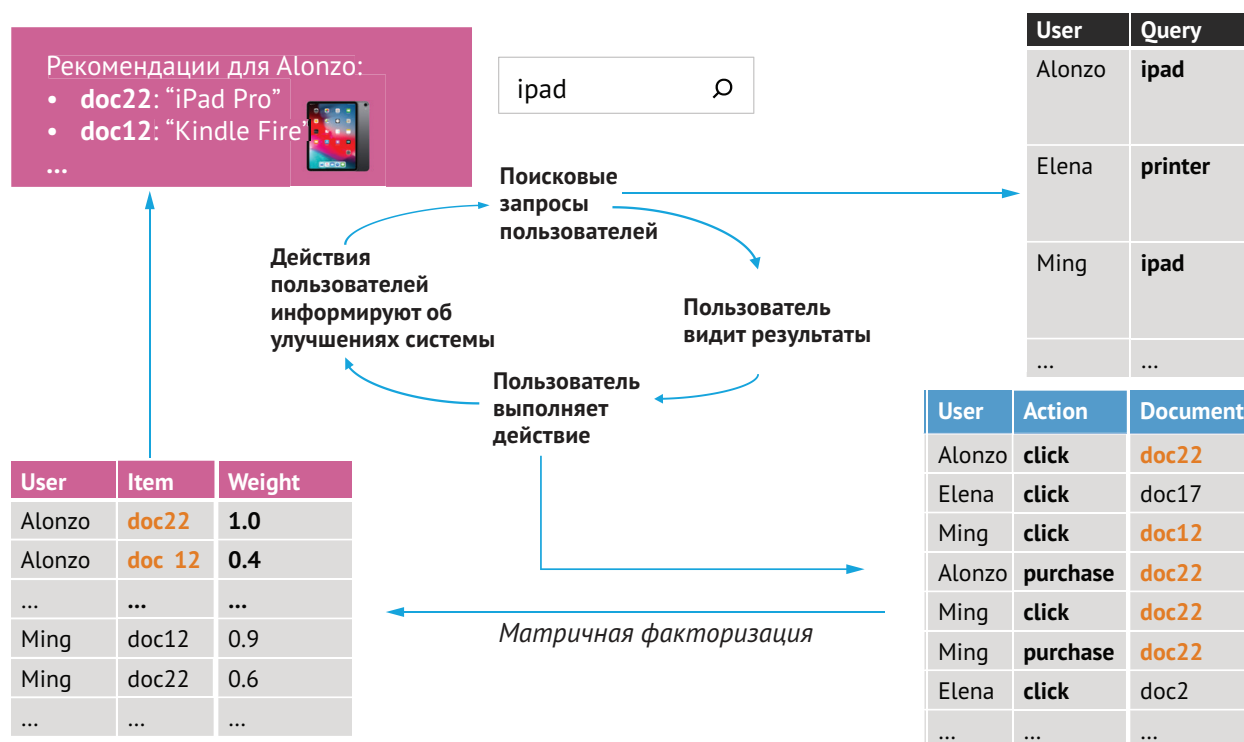


Рис. 4.6. Совместная фильтрация для рекомендаций пользователя к предмету. Основываясь на своем прошлом поведении, наш пользователь (Алонзо) получает рекомендации, основанные на предметах, которые понравились другим пользователям, причем эти пользователи также взаимодействовали с некоторыми из тех же предметов, что и Алонзо

Как и бустинг сигналов, совместная фильтрация включает в себя непрерывный цикл обратной связи. Сигналы собираются, по этим сигналам строятся модели, этими моделями генерируются рекомендации, а взаимодействия с этими рекомендациями затем снова регистрируются как дополнительные сигналы. Подходы к совместной фильтрации обычно генерируют матрицу взаимодействия пользователя с предметом, сопоставляя каждого пользователя с каждым предметом (документом), при этом сила связи между каждым пользователем и предметом основывается на силе положительных взаимодействий (клики, покупки, оценки и т. д.).

Если матрица взаимодействия достаточно заполнена, из нее можно вывести рекомендации для любого пользователя или предмета с данными о взаимодействии. Это делается путем прямого поиска других пользователей, которые взаимодействовали с тем же предметом, а затем бустинга других предметов (аналогично бустингу сигналов), с которыми эти пользователи также взаимодействовали. Однако, если матрица взаимодействия пользователя с предметом слишком разрежена, обычно необходимо применять подход факторизации матрицы.

Факторизация матрицы – это процесс разбиения матрицы взаимодействия пользователь–предмет на две матрицы: одна сопоставляет пользователей со скрытыми признаками (или *факторами*), а другая сопоставляет эти скрытые факторы с предметами. Это похоже на подход к снижению размерности, который мы описали в главе 3, где мы перешли от представления продуктов питания с множеством точных ключевых слов (вектор, включающий признак для каждого слова в инвертированном индексе) к использованию гораздо меньшего количества значимых измерений (вектор с восемью признаками, описывающими продукты питания), в которые мы сжали данные. Эта матричная факторизация позволяет выводить предпочтения пользователя в отношении предметов, а также сходство между предметами путем преобразования ограниченных данных сигналов в меньшее количество более значимых измерений, которые лучше обобщают сходство между предметами.

В контексте матричной факторизации для совместной фильтрации скрытые факторы представляют атрибуты наших документов, которые, как известно, являются важными индикаторами общих интересов пользователей. Сопоставляя другие документы на основе этих факторов, мы используем краудсорсинг для поиска других похожих документов, соответствующих тем же общим интересам.

Насколько бы эффективной ни была совместная фильтрация для изучения интересов и вкусов пользователей, основанная исключительно на краудсорсинговой релевантности, она страдает от серьезного недостатка, известного как *проблема холодного старта*. Это сценарий, в котором возврат результатов зависит от наличия сигналов, но в котором новые документы, которые никогда не генерировали сигналы, не будут возвращены. Это создает ситуацию

«уловки-22»¹, когда новый контент вряд ли будет показан пользователям (предпосылка для генерации сигналов), потому что он еще не сгенерировал никаких сигналов (которые необходимы для показа контента). В некоторой степени модели бустинга сигналов демонстрируют похожую проблему, когда документы, которые уже популярны, как правило, получают более высокий бустинг, в результате чего они получают еще больше сигналов, в то время как невидимые документы продолжают не получать бустинга сигналов вообще. Этот процесс создает самоусиливающийся цикл, который может привести к отсутствию разнообразия в результатах поиска. Эта проблема называется *предвзятостью представления*, и мы покажем, как ее преодолеть, в главе 12.

Вы также можете генерировать рекомендации другими способами, например с помощью рекомендаций на основе контента, что мы рассмотрим в следующей главе (раздел 5.4.6). Однако совместная фильтрация уникальна тем, что она может изучать предпочтения и вкусы пользователей в отношении других документов, не имея никаких знаний об их содержании. Это связано с тем, что все решения принимаются исключительно путем наблюдения за взаимодействием пользователей с контентом и определения степени сходства на основе этих наблюдений. Мы более подробно рассмотрим совместную фильтрацию при реализации персонализированного поиска в главе 9.

Вместо того чтобы использовать только популярные и персонализированные модели релевантности (которые лучше всего работают, когда документы уже имеют сигналы), поисковая система может также извлечь выгоду из более обобщенной модели релевантности, которая может применяться ко всем поискам и документам. Это во многом способствует решению проблемы холодного старта. Далее мы рассмотрим, как краудсорсинговая релевантность может быть обобщена с помощью техники, называемой «обучение ранжированию».

4.2.4. **Обобщенная релевантность через обучение ранжированию**

Поскольку бустинг сигналов (популяризированная релевантность, раздел 4.2.2) и совместная фильтрация (персонализированная релевантность, раздел 4.2.3) работают только с документами, которые уже имеют сигналы, значительная часть запросов не получит выгоды, пока документы не получат трафик. Именно здесь обучение ранжированию оказывается ценным как форма обобщенной релевантности.

Обучение ранжированию (LTR), также известное как *ранжирование с машинным обучением*, представляет собой процесс создания и исполь-

¹ «Уловка-22» – роман американского писателя Джозефа Хеллера, опубликованный в 1961 году. Известен возникшим в нем логическим парадоксом между взаимоисключающими правилами. Термин «уловка-22» обозначает парадоксальную и безвыходную ситуацию, когда для выполнения одного предписания нужно нарушить другое, что категорически запрещено правилами. – *Прим. ред.*

зования классификатора ранжирования, который может оценить, насколько хорошо любой документ соответствует любому произвольному запросу. Вы можете представить себе классификатор ранжирования как обученную модель релевантности. Вместо ручной настройки поисковых бустингов и других параметров процесс LTR тренирует модель машинного обучения, которая может понимать важные особенности ваших документов, а затем соответствующим образом оценивать результаты поиска. На рис. 4.7 показан общий поток для развертывания LTR.

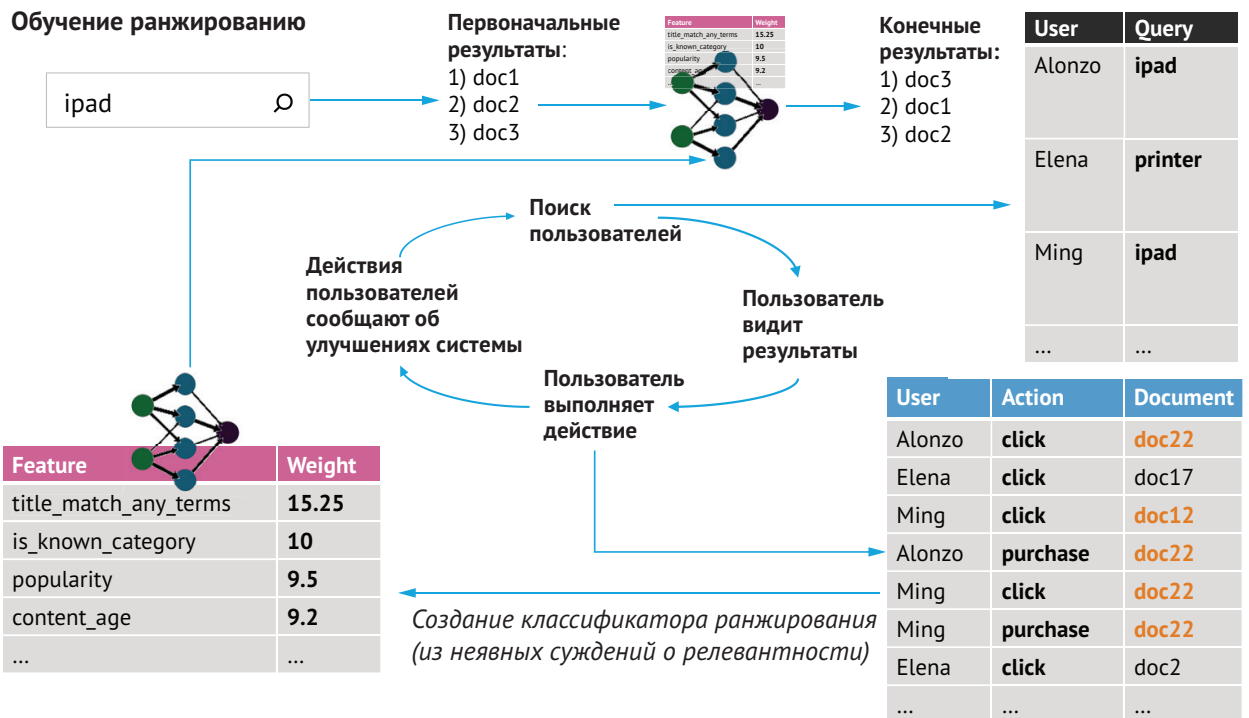


Рис. 4.7. Обучение ранжированию (обобщенная релевантность). Классификатор ранжирования строится на основе суждений пользователей об известной релевантности документов для каждого запроса (обучающий набор). Затем эта модель классификатора ранжирования используется для реранкинга результатов поиска, чтобы документы с самым высоким рейтингом были более релевантными

В системе LTR применяется тот же высокоуровневый процесс отраженного интеллекта, что и в бустинге сигналов и совместной фильтрации (см. рис. 4.2). Разница в том, что LTR может использовать списки суждений о релевантности (карты запросов с их идеальным ранжированным набором документов) для автоматического обучения модели релевантности, которая затем может применяться в целом ко всем запросам. Вы увидите, что вывод шага «Создание классификатора ранжирования» на рис. 4.7 представляет собой модель функций релевантности (title_match_any_terms, is_known_category, popularity, и content_age), и эта модель периодически развертывается в рабочей поисковой системе для улучшения ранжирования результатов поиска. Функции в очень простой модели ранжирования с машинным обучением могут быть читаемыми, как эта, но нет требования, чтобы классификатор ранжирования был интерпретируемым или объяснимым,

и многие продвинутые классификаторы ранжирования на основе глубокого обучения таковыми не являются.

На рис. 4.7 обратите внимание, что поток живого пользователя начинается с запроса для iPad. Затем первоначальные результаты поиска пропускаются через развернутый классификатор обучения ранжированию, который возвращает окончательный переранжированный набор результатов поиска. Поскольку классификатор ранжирования, как правило, гораздо более интеллектуален и использует более сложные параметры ранжирования, чем традиционная модель релевантности ранжирования ключевых слов, обычно это слишком долгий путь – использовать классификатор ранжирования для оценки всех соответствующих документов в поисковой системе. Вместо этого LTR часто использует начальную, более быструю функцию ранжирования (например, BM25) для поиска топ-N документов (обычно сотни или тысячи документов), а затем только пропускает это подмножество документов через классификатор ранжирования. Можно использовать классификатор ранжирования в качестве основной функции релевантности вместо применения этой техники реранкинга, но чаще можно увидеть подход реранкинга, так как он, как правило, намного быстрее, и к тому же дает похожие результаты.

LTR может использовать либо явные суждения о релевантности (создаваемые вручную экспертами), либо неявные суждения (выведенные из сигналов пользователя), или некоторую комбинацию этих двух. Мы рассмотрим примеры реализации LTR из явных и неявных списков суждений в главах 10–12.

4.2.5. Другие модели отраженного интеллекта

Помимо более глубокого погружения в бустинг сигналов (глава 8), совместную фильтрацию (глава 9) и обучение ранжированию (глава 10), мы рассмотрим многие другие виды отраженного интеллекта в этой книге. В главе 6 мы рассмотрим интеллектуальный анализ пользовательских запросов для автоматического изучения домен-специфичных фраз, распространенных орфографических ошибок, синонимов и связанных терминов, а в главах 11–12 мы рассмотрим автоматизированные способы изучения суждений о релевантности из взаимодействий пользователей для автоматической генерации обучающих данных для новых подходов к машинному обучению.

В целом каждое взаимодействие между пользователем и контентом создает связь – ребро в графе, которое мы можем использовать для понимания возникающих отношений и получения более глубоких знаний. На рис. 4.8 показаны некоторые из различных отношений, которые мы можем изучить, исследуя этот граф взаимодействия. Одни и те же входящие данные сигналов могут обрабатываться по-разному, с помощью различных подходов к агрегации сигналов и машинному обучению, чтобы определить:

- сходство между пользователями и предметами (рекомендации пользователь–предмет);

- сходство между предметами и другими предметами (рекомендации предмет–предмет);
- специфические предпочтения на основе признаков, которые могут генерировать профиль интересов пользователя;
- сходство между запросами и предметами.

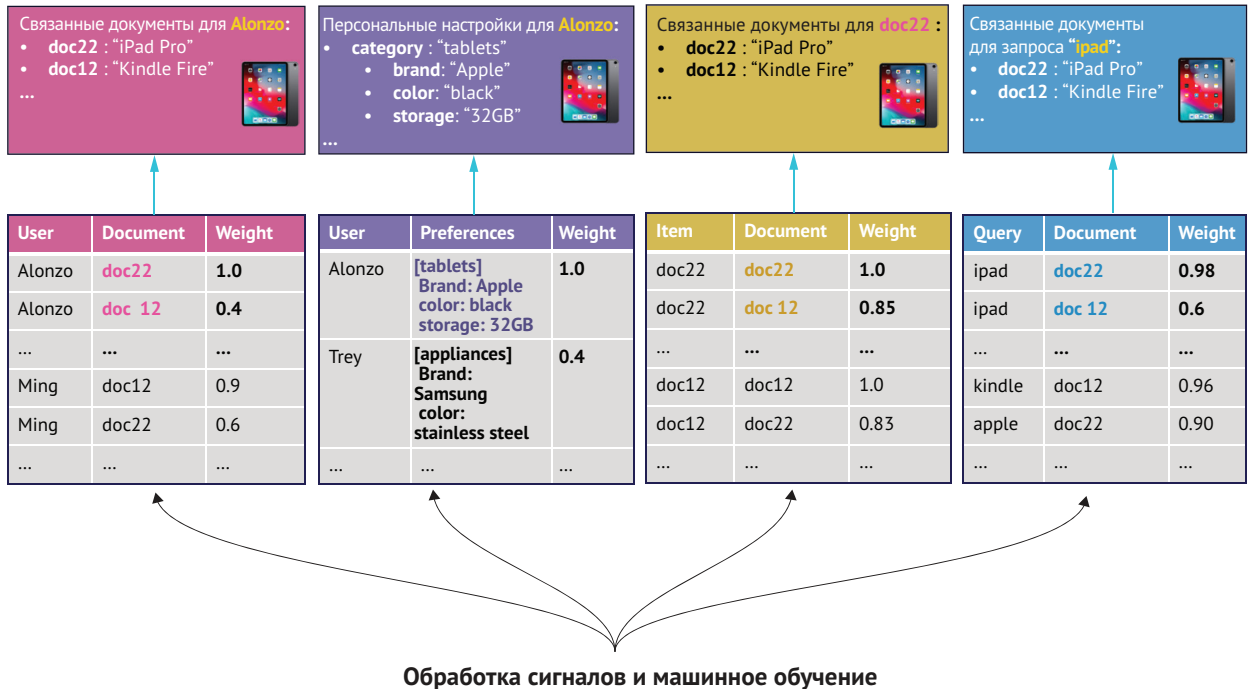


Рис. 4.8. Множество отраженных моделей интеллекта. Крайний левый блок отображает сходство пользователя с предметом для рекомендаций, следующий блок отображает изучение предпочтений на основе определенных признаков для профиля пользователя, третий блок отображает изучение сходства предмета с предметом для рекомендаций, а крайний правый блок отображает изучение запроса с предметом

Мы продолжим изучать эти методы в следующих главах, но надо иметь в виду, что данные сигналов содержат сокровищницу потенциальных идей и часто дают столько же преимуществ, сколько и содержимое документов, участвующих во взаимодействиях с пользователем. Отраженный интеллект и краудсорсинг не ограничиваются только бустингом сигналов, совместной фильтрацией и методами обучения ранжированию, которые мы описали. Они также могут быть получены из контента вместо сигналов, что мы обсудим в следующем разделе.

4.2.6. Краудсорсинг из контента

Хотя мы обычно думаем о краудсорсинге как о просьбе пользователей предоставить входные данные, мы увидели в этой главе, что неявная обратная связь часто может обеспечить такую же или даже большую ценность в совокупности по многим пользовательским сигналам. Хотя эта глава была полностью сосредоточена на использовании пользовательских сигналов для выполнения этого краудсорсинга, также важно отметить, что сам контент может использоваться в качестве краудсорсингового интеллекта для вашей поисковой платформы на базе ИИ.

Например, если вы пытаетесь выяснить общее качество ваших документов, вы можете просмотреть отзывы клиентов, чтобы создать рейтинг продукта или узнать, был ли он отмечен как оскорбительный или спам. Если клиент оставил комментарии, вы можете запустить алгоритм *анализа настроений*¹ в тексте, обозначив, являются ли комментарии положительными, нейтральными или отрицательными. На основе обнаруженного настроения вы можете повысить или понизить исходные документы соответственно. Этот процесс, по сути, извлекает сигналы из контента, отправленного пользователями, поэтому это все еще форма краудсорсинга, хотя и из другого контента, предоставленного пользователями.

Мы упоминали, что в главе 6 будет рассмотрено, как можно добывать сигналы пользователей, чтобы автоматически изучать домен-специфичную терминологию (фразы, орфографические ошибки, синонимы и т. д.). Так же как вы можете использовать запросы и взаимодействия пользователей для изучения этой терминологии, вы должны понимать, что документы, как правило, пишутся людьми и что, следовательно, очень похожие виды отношений между терминами отражаются в написанном контенте. Мы рассмотрим эти отношения на основе контента подробнее в следующей главе.

Одним из самых известных существующих алгоритмов поиска является алгоритм *Page Rank* – прорывной алгоритм, который изначально позволил Google занять видное место как наиболее релевантной поисковой системе в интернете. Page Rank выходит за рамки текста любой данной веб-страницы и рассматривает подразумеваемое поведение всех других создателей веб-страниц, чтобы увидеть, как они ссылались на другие веб-страницы. Измеряя входящие и исходящие ссылки, можно измерить «качество» веб-страницы, предполагая, что веб-сайты с большей вероятностью ссылаются на более качественные, более авторитетные источники и что эти более качественные источники с меньшей вероятностью ссылаются на источники более низкого качества. Эта идея выхода за рамки контента в одном документе и связывания его с другими документами – будь то прямые ссылки между ними, комментарии или отзывы пользователей, любые другие взаимодействия пользователей или даже просто использование терминологии разными, нюансированными способами в документах – невероятно эффективна. Искусство и наука использования всей доступной информации о вашем контенте и от ваших пользователей являются ключом к созданию высокорелевантной поисковой системы на базе ИИ. В главе 5 мы рассмотрим графы знаний и то, как мы можем использовать некоторые из отношений, встроенных в подразумеваемые ссылки между документами, для автоматического дальнейшего понимания предметной области.

¹ Анализ настроений (англ. *sentiment analysis*) в программировании – это вычислительный метод, который позволяет определить и категоризировать эмоциональный тон или отношение, выраженное во фрагменте текста. – *Прим. ред.*

Резюме

- Контент, сигналы и модели (которые выводятся из контента и сигналов) являются тремя основными источниками «топлива» для работы поисковой системы на базе ИИ, причем сигналы являются основным источником для краудсорсинговой релевантности.
- Отраженный интеллект – это процесс создания циклов обратной связи обучения, которые улучшаются с каждым взаимодействием пользователя и отражают этот усвоенный интеллект обратно, чтобы постоянно повышать релевантность будущих результатов.
- Бустинг сигналов – это форма популяризированной релевантности, которая обычно оказывает наибольшее влияние на ваши самые рейтинговые, самые популярные запросы.
- Совместная фильтрация – это форма персонализированной релевантности, которая может использовать шаблоны взаимодействия пользователя с предметами для изучения предпочтений пользователя или силы связей между предметами, а затем рекомендовать похожие предметы на основе этих изученных связей.
- Обучение ранжированию (LTR) – это форма обобщенной релевантности, которая представляет собой процесс обучения классификатора ранжирования на основе списков суждений о релевантности (сопоставление запросов с правильно ранжированными документами). LTR можно применять для ранжирования всех документов и таким образом избегать проблемы холодного старта.
- Существуют и другие виды отраженного интеллекта, включая методы использования контента (а не только сигналов) для краудсорсинговой релевантности.

Часть 2

Изучение домен-специфического намерения

В части 1 вы изучили механику сопоставления и ранжирования по ключевым словам (с использованием TF-IDF и BM25) и числовым векторам (с использованием косинуса или скалярного произведения). Вы также познакомились с обзором краудсорсингового ранжирования релевантности. Однако перед освоением этих методов ранжирования важно научиться правильно интерпретировать запрос пользователя и выполнять соответствующий поиск, который понимает намерение пользователя. Если запущен неверный запрос и сопоставлены неверные документы, никакая логика ранжирования этих плохих результатов не сможет преодолеть ошибку неверно истолкованного запроса.

Правильная интерпретация запроса пользователя – на этом сфокусирована часть 2. Этот шаг часто упускается из виду, поскольку он очень специфичен для домена, но тем не менее он имеет решающее значение для удовлетворения информационных потребностей пользователя. В главе 5 вы научитесь проходить по семантическим графам знаний, гигантским графам гиперструктурированных данных внутри вашей поисковой системы, которые были представлены в главе 2. Эти графы лежат в основе поисковых систем и позволяют делать выводы о тонком значении домен-специфичных терминов в ваших данных. В главе 6 вы научитесь выполнять классификацию намерений запросов, используя эти графы для устранения неоднозначности смысла

слов и фраз, и как использовать и ваш контент, и сигналы поведения пользователей для изучения домен-специфичной терминологии, связанных терминов, опечаток и альтернативных форм терминов. В главе 7 мы свяжем все, что вы узнали, вместе и построим конвейер запросов для выполнения семантического поиска. Этот конвейер будет анализировать домен-специфичные намерения запросов ваших пользователей и переписывать их, чтобы представлять поисковой системе более точную семантическую интерпретацию намерения пользователя, что приведет к гораздо более релевантным результатам поиска.

5

Что такое графы знаний

В этой главе рассматривается:

- создание графов знаний и работа с ними;
- реализация открытого извлечения информации для создания графов знаний из текста;
- обнаружение произвольных семантических связей с помощью семантических графов знаний;
- расширение и переписывание запросов с использованием графов знаний;
- интерпретация документов с помощью графов знаний.

В предыдущей главе мы в первую очередь сосредоточились на изучении сходства между запросами и документами на основе поведенческих сигналов пользователей. В главе 2 мы также обсудили, как текстовый контент документа, вместо того чтобы быть неструктурированными данными, больше похож на гигантский граф гиперструктурированных данных, содержащий богатый граф семантических связей, связывающих множество последовательностей символов, терминов и фраз, которые существуют в наших коллекциях документов.

В этой главе мы покажем, как использовать этот гигантский граф семантических связей в нашем контенте для лучшей интерпретации домен-специфичной терминологии. Мы достигнем этого, используя как традиционные графы знаний, которые позволяют явно модели-

ровать отношения в пределах домена, так и семантические графы знаний, которые позволяют в реальном времени выводить тонкие семантические отношения в пределах домена.

Семантический граф знаний – это простой вид *языковой модели* (языковая модель представляет собой распределение вероятностей по последовательностям слов). Мы будем использовать семантический граф знаний как ступеньку к пониманию больших языковых моделей (LLM) в последующих главах. LLM – это глубокие нейронные сети, обычно обучаемые на миллиардах параметров и огромных объемах данных (часто большей части известного интернета) для моделирования общего представления человеческих знаний. Однако семантические графы знаний – это запрашиваемые языковые модели, представляющие только те отношения, которые фактически находятся в вашем поисковом индексе. Хотя семантические графы знаний не содержат возможности рассуждать в целом о языке, они могут быть очень мощными для домен-специфичного контекстного вывода, как мы увидим.

В этой главе мы также поработаем с несколькими интересными наборами данных, чтобы продемонстрировать разнообразие способов построения и применения графов знаний для улучшения понимания запросов в различных областях.

5.1. Работа с графами знаний

В разделе 2.4 мы представили идею *графов знаний* и обсудили, как они соотносятся с другими типами моделей знаний, такими как онтологии, таксономии, синонимы и альтернативные метки. Графы знаний, если вы помните, объединяют все эти типы моделей знаний, поэтому мы будем называть их общим термином «графы знаний», когда будем строить их на протяжении всей этой главы.

Граф знаний (или любой граф, если на то пошло) определен с помощью понятия узлов (также известных как *вершины*) и ребер. *Узел* – это сущность, представленная в графе знаний (например, термин, человек, место, вещь или понятие), тогда как *ребро* представляет собой связь между двумя узлами. На рис. 5.1 показан пример графа, отображающего узлы и ребра.

На этом рисунке вы можете видеть четыре узла, представляющих авторов, один узел, представляющий исследовательскую работу, которую они написали вместе, один узел, представляющий научную конференцию, на которой работа была представлена и опубликована, а затем узлы, представляющие город, провинцию, страну и даты, в течение которых проводилась конференция. Проходя (или «следуя») по независимым ребрам между узлами, вы можете сделать вывод, что один из авторов был в Монреале, Канада, в октябре 2016 года. Хотя любая структура с узлами и ребрами, как эта, считается графом, этот конкретный граф представляет фактические знания и, следовательно, также считается графом знаний.

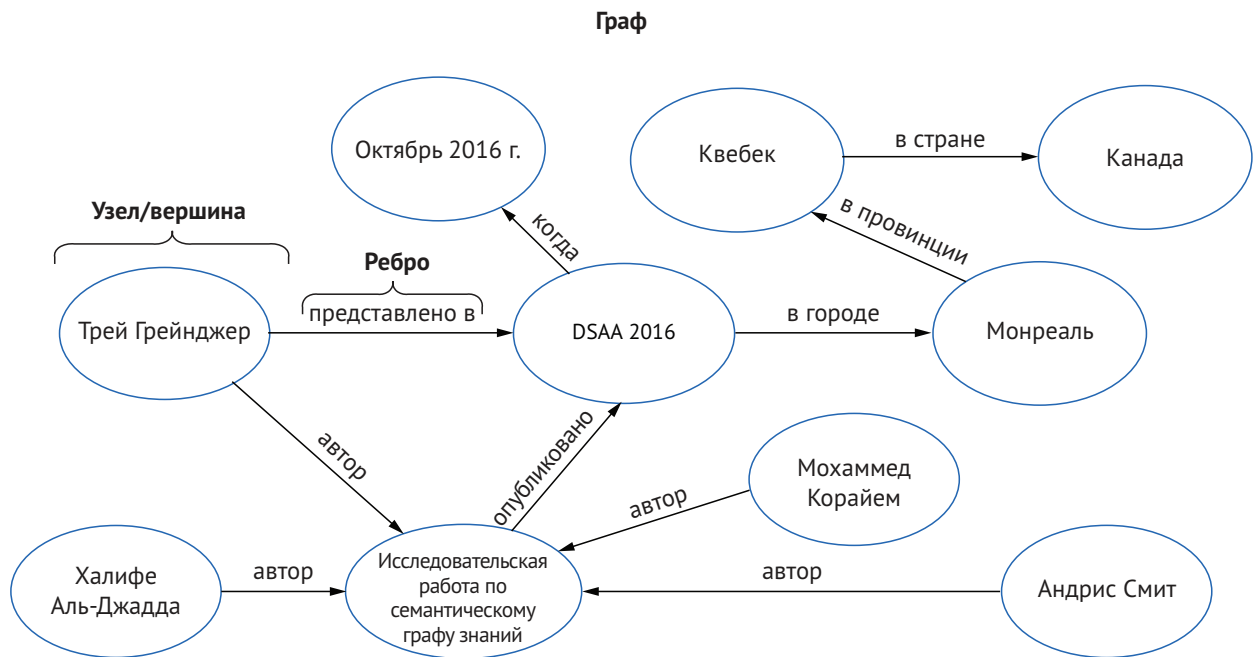


Рис. 5.1. Структура графа. Графы состоят из узлов (также называемых «вершины»), которые представляют сущности, и ребер, которые представляют связь одного узла графа узла с другим узлом. Графы предоставляют способ построения моделей знаний и вывода новых идей путем обхода ребер (или «следования по ребрам») от одного узла к другому

Существует множество способов построения и представления графов знаний, как посредством явного моделирования данных в виде узлов и ребер, так и посредством динамической материализации (обнаружения) узлов и ребер из ваших данных в реальном времени. Последнее известно как *семантический граф знаний*. В этой главе мы рассмотрим различные примеры, включая построение явного графа знаний вручную, автоматическую генерацию явного графа знаний и использование семантического графа знаний, который уже присутствует в вашем поисковом индексе. Чтобы начать работу с графами знаний, у вас есть, по сути, три варианта:

- построить граф знаний с нуля, используя графовую базу данных (Neo4j, Apache TinkerPop, ArangoDB и т. д.);
- подключить уже существующий граф знаний (ConceptNet, DBpedia, большую языковую модель и т. д.);
- автоматически сгенерировать граф знаний из ваших данных, используя ваш контент напрямую для извлечения знаний.

Каждый подход имеет свои сильные и слабые стороны, хотя подходы не обязательно являются взаимоисключающими. Если вы создаете поисковую систему общего знания (например, поисковую систему в интернете), использование уже существующего графа знаний или большой языковой модели – это отличное место для начала. Однако, если ваша поисковая система более домен-специфична, ваши домен-специфичные сущности и терминология могут не присутствовать в уже существующем графе, что потребует от вас создания пользовательского графа знаний. В этой главе мы сосредоточимся в первую очередь на третьем варианте: автоматическое создание графа знаний из вашего кон-

тента. Другие два метода уже хорошо описаны во внешних материалах с использованием таких технологий, как SPARQL, RDF Triples и Apache Jena, или уже существующих графов знаний, таких как DBpedia и Yago. Вам все равно нужно будет иметь возможность переопределять свой граф знаний и добавлять пользовательский контент, поэтому мы включим примеры интеграции как явно определенных графов знаний (построенные со списком предопределенных связей), так и неявно определенных графов знаний (автоматически созданные связи, обнаруженные динамически из данных) в вашу поисковую платформу.

5.2. Использование нашей поисковой системы в качестве графа знаний

Многие организации тратят значительные ресурсы на создание графов знаний для своих организаций, но испытывают трудности с их интеграцией в свои поисковые системы. К счастью, для наших примеров мы выбрали реализацию поисковой системы по умолчанию (Apache Solr), которая имеет встроенные явные возможности обхода графа, поэтому нет необходимости в подключении новой внешней системы для реализации или обхода наших графов знаний.

Хотя могут быть некоторые преимущества использования внешней базы данных графов, такой как Neo4J или ArangoDB, которая поддерживает более сложную семантику обхода графа, использование внешней системы, подобной этой, усложняет координацию запросов, синхронизацию данных и управление инфраструктурой. Кроме того, поскольку некоторые виды операций с графами могут эффективно выполняться только в поисковой системе (например, обходы семантического графа знаний с использованием инвертированного индекса, с которым мы вскоре столкнемся), использование поисковой системы в качестве единой платформы для возможностей поиска и графа знаний сокращает количество систем, которыми нам нужно будет управлять.

Мы подробно рассмотрим реализацию системы семантического поиска в главе 7, включая семантический анализ запросов, извлечение фраз, обнаружение орфографических ошибок, расширение синонимов и переписывание запросов, все из которых будут включены в явно построенный граф знаний. Поскольку цель текущей главы – сосредоточиться на *изучении* графа знаний, мы отложим большую часть обсуждения шаблонов интеграции во время запроса до главы 7, когда сможем связать все из этой главы и главы 6 в соответствующую структуру графа знаний.

5.3. Автоматическое извлечение графов знаний из контента

Хотя вам нужно будет иметь возможность изменять узлы и ребра в ваших графах знаний, ручное поддержание крупномасштабного графа знаний очень сложно. Поддерживаемые вручную графы знаний требуют суще-

ственного предметного опыта, должны активно обновляться с учетом изменяющейся информации и подвержены предвзятостям и ошибкам тех, кто их поддерживает.

Извлечение открытой информации – это развивающаяся область исследований обработки естественного языка (NLP). Извлечение открытой информации направлено на извлечение фактов непосредственно из вашего текстового контента. Это часто делается с использованием библиотек NLP и языковых моделей для разбора предложений и оценки графа зависимости между ними. Граф зависимости представляет собой разбивку частей речи для каждого слова и фразы в предложении, а также указание того, какие слова ссылаются на какие другие слова. Более поздние подходы к извлечению графа знаний, как правило, используют LLM, специально обученные для извлечения сущностей, такие как UniRel (унифицированное представление и взаимодействие для совместного реляционного тройного извлечения) и REBEL (извлечение отношений путем сквозной языковой генерации). Подходы на основе LLM, вероятно, со временем станут стандартом для извлечения графа знаний из-за их способности представлять и извлекать более тонкие отношения между сущностями, чем традиционные подходы на основе графа зависимости. Однако в целях обучения в этой главе мы сосредоточимся на подходе на основе графа зависимости, поскольку он обеспечит лучшую основу для понимания механики извлечения графа знаний из текста и возможности создания пользовательских шаблонов извлечения отношений. Вы всегда можете переключиться на более продвинутый подход, основанный на LLM, позже, если он лучше соответствует вашим потребностям.

В этом разделе мы будем использовать языковую модель и графы зависимостей для извлечения двух различных типов отношений: произвольных отношений и отношений гипонимов¹.

5.3.1. Извлечение произвольных отношений из текста

Учитывая гиперструктурированную природу текста и богатые отношения, выраженные в типичных предложениях и абзацах, само собой разумеется, что мы должны иметь возможность идентифицировать субъекты и объекты предложений и то, как они связаны. В этом разделе мы сосредоточимся на извлечении произвольных отношений между сущностями, описанными в предложениях нашего текстового контента.

Анализируя существительные и глаголы в предложении, часто можно вывести факт, который присутствует в предложении, и сопоставить этот факт с триплетом RDF (также известным как семантический триплет). Структура описания ресурсов (RDF) – это модель данных, используемая для представления графиков и отношений. *Триплет* RDF – это трехкомпонентная структура данных, представляющая субъект (началь-

¹ Гипоним – понятие, выражающее частную сущность по отношению к другому, более общему понятию. Другими словами, это слово с более узким значением. Пример: термин «бульдог» – гипоним по отношению к термину «собака», а «собака» – гипоним по отношению к термину «зверь». – *Прим. ред.*

ный узел), связь (ребро) и объект (конечный узел). Например, в предложении «Колин посещает среднюю школу Риверсайд» глагол «посещает» может быть извлечен как тип связи, соединяющий субъект («Колин») с объектом («Средняя школа Риверсайд»). Таким образом, триплет RDF выглядит следующим образом: ("Colin", "attends", "Riverside High School").

В листинге 5.1 представлен пример использования библиотеки spaCy на основе Python для извлечения фактов из текстового контента. SpaCy – популярная библиотека обработки естественного языка, которая поставляется с современными статистическими моделями нейронных сетей для разметки частей речи, анализа зависимостей, категоризации текста и распознавания именованных сущностей.

Листинг 5.1. Извлечение связей и разрешение кореферентностей¹

```

def extract_relationships(text, lang_model, coref_model):
    resolved_text = resolve_coreferences(text, coref_model)
    sentences = get_sentences(resolved_text, lang_model)
    return resolve_facts(sentences, lang_model)

text = """
Data Scientists build machine learning models. They also write code.
Companies employ Data Scientists.
Software Engineers also write code. Companies employ Software Engineers.
"""

lang_model = spacy.load("en_core_web_sm")
coref_model = spacy.load("en_coreference_web_trf")
graph = extract_relationships(text, lang_model, coref_model)
print(graph)

Вывод:

sentence: Data Scientists build machine learning models.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
-----
sentence: Data Scientists also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
-----
sentence: Companies employ Data Scientists.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
-----
sentence: Software Engineers also write code.
dependence_parse: ['nsubj', 'advmod', 'ROOT', 'dobj', 'punct']
-----
sentence: Companies employ Software Engineers.
dependence_parse: ['nsubj', 'ROOT', 'dobj', 'punct']
-----

```

Разрешает сущности, например заменяет местоимения существительными.

Классифицирует части речи для текста.

Генерирует триплеты RDF.

Экспериментальная модель spaCy, используемая для разрешения кореферентов.

¹ Кореферентность (референциональное тождество) – отношение между именами – компонентами высказывания, в котором имена ссылаются на один и тот же объект (ситуацию) внеязыковой действительности (референт). – *Прим ред.*

```
[['Data Scientists', 'build', 'machine learning models'],  
 ['Data Scientists', 'write', 'code'],  
 ['Companies', 'employ', 'Data Scientists'],  
 ['Software Engineers', 'write', 'code'],  
 ['Companies', 'employ', 'Software Engineers']]
```

Как вы понимаете, пример кода взял текстовое содержимое, разбил его на предложения, а затем определил субъекты, отношения и объекты в этих предложениях. Затем эти триплеты RDF можно сохранить в явно построенный граф знаний и обойти его.

Рисунок 5.2 представляет визуализацию этого извлеченного графа. Хотя этот пример является базовым, продвинутые алгоритмы могут извлекать факты из более сложных лингвистических шаблонов. Мы используем библиотеку spaCy в примере кода, которая использует нейронную языковую модель на основе глубокого обучения для обнаружения частей речи, фраз, зависимостей и кореферентностей во входном тексте. Механизм, который мы затем применяем для разбора этих лингвистических выходов в триплеты RDF, больше основан на правилах, следуя известным семантическим шаблонам в английском языке.

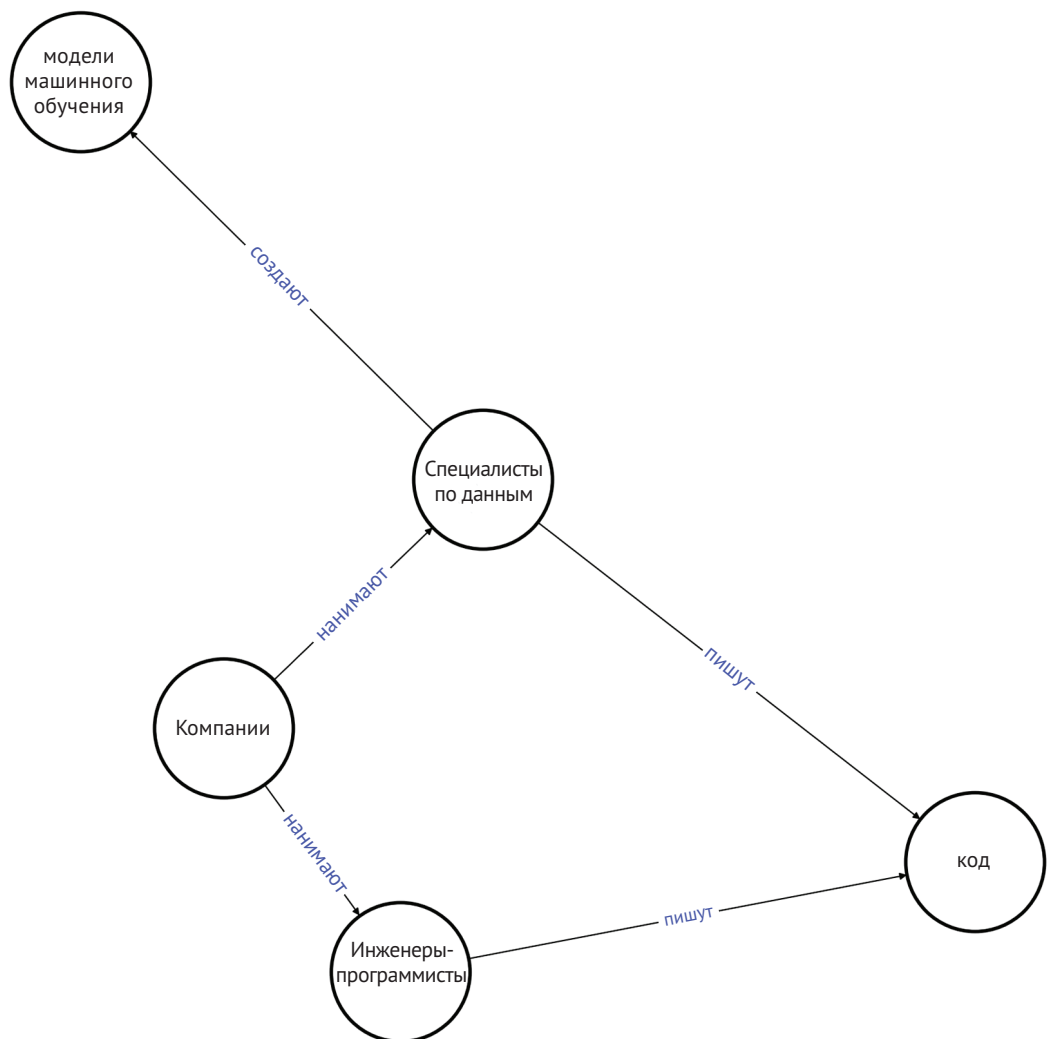


Рис. 5.2. Извлеченный граф знаний. Узлы и ребра в этом графе были автоматически извлечены из текстового контента на основе шаблонов частей речи

К сожалению, при парсинге произвольных глаголов в отношениях таким образом извлеченные отношения могут стать довольно шумными. Поскольку глаголы спрягаются по-разному, имеют синонимы и перекрывающиеся значения, часто необходимо сокращать, объединять и иным образом очищать любой список произвольно извлеченных связей.

Напротив, некоторые типы связей намного проще, например статистические связи («связан с») и гипонимы («является»). Мы посвятим оставшуюся часть главы, сосредоточившись в первую очередь на использовании этих двух особых типов, начав с гипонимов.

5.3.2. Извлечение гипонимов и гиперонимов из текста

Хотя сопоставление произвольных глаголов для чистых списков отношений в графе знаний может быть сложной задачей, извлечение гипонимов и гиперонимов может быть намного проще. *Гипонимы* – это сущности, которые поддерживают связь «is a» или «is instance of» с более общей формой сущностей, причем более общая форма называется *гиперонимом*¹. Например, для связей между терминами «phillips head», «screwdriver» и «tool» мы бы сказали, что «phillips head» является гипонимом «screwdriver», «tool» – гиперонимом «screwdriver» и «screwdriver» является как гиперонимом «phillips head», так и гипонимом «tool».

Один из распространенных и довольно точных способов извлечения связей гипоним/гипероним из текста – использование шаблонов Херста, описанных Марти Херстом в «Автоматическом извлечении гипонимов из больших текстовых корпусов» (в *COLING 1992 Том 2: 14-я Международная конференция по компьютерной лингвистике, 1992*). Эти шаблоны описывают общие лингвистические шаблоны, которые надежно указывают на наличие гипонимов в предложениях. Следующий листинг демонстрирует несколько примеров таких шаблонов.

Листинг 5.2. Шаблоны Херста, которые идентифицируют семантические отношения

```
simple_hearst_patterns = [
    ("(NP_\\w+ (, )?such as (NP_\\w+ ?(, )?(and |or )?)+)", "first"),
    ("(such NP_\\w+ (, )?as (NP_\\w+ ?(, )?(and |or )?)+)", "first"),
    ("((NP_\\w+ ?(, )?)+(and |or )?other NP_\\w+)", "last"),
    ("(NP_\\w+ (, )?include (NP_\\w+ ?(, )?(and |or )?)+)", "first"),
    ("(NP_\\w+ (, )?especially (NP_\\w+ ?(, )?(and |or )?)+)", "first")]
```

¹ Гипероним – слово с более широким значением, выражающее общее, родовое понятие, название класса (множества) предметов (свойств, признаков). В лингвистике – понятие в отношении к другому понятию, выражающее более общую сущность. В отношении некоторого множества объектов – понятие, отражающее надмножество к исходному. Пример: термин «животное» является гиперонимом по отношению к термину «собака», а термин «собака» в свою очередь – гипероним по отношению к термину «бульдог». Гипероним является результатом логической операции обобщения, тогда как гипоним – ограничения. – *Прим. ред.*

Каждый из этих пяти простых шаблонов представлен в виде кортежа Python¹, где первая запись является *регулярным выражением*, а вторая – позицией в сопоставлении шаблона (т. е. *first* или *last*). Если вы не знакомы с регулярными выражениями, они предоставляют общий и мощный синтаксис для сопоставления шаблонов в строках. Везде, где вы видите символы *NP*, это означает наличие *именной фразы* (noun phrase) в предложении. Позиция, указанная во втором элементе кортежа (*first* или *last*), указывает, какая именно группа в предложении представляет гипероним, а все остальные именные группы, соответствующие шаблону, считаются гипонимами.

В следующем листинге мы выполняем почти 50 из этих шаблонов Hearst, чтобы сопоставить множество комбинаций отношений «is a» в нашем контенте.

Листинг 5.3. Извлечение отношений гипонима с использованием шаблонов Херста

```
text_content = """Many data scientists have skills such as machine
learning, python, deep learning, apache spark, among others. Job candidates
most prefer job benefits such as commute time, company culture, and salary.
Google, Apple, or other tech companies might sponsor the conference.
Big cities such as San Francisco, Miami, and New York often appeal to
new graduates. Job roles such as Software Engineer, Registered Nurse,
and DevOps Engineer are in high demand. There are job benefits including
health insurance and pto."""
```

```
extracted_relationships = HearstPatterns().find_hyponyms(text_content)
facts = [[pair[0], "is_a", pair[1]] for pair in extracted_relationships]
print(*facts, sep="\n")
```

Вывод:

```
['machine learning', 'is_a', 'skill']
['python', 'is_a', 'skill']
['deep learning', 'is_a', 'skill']
['apache spark', 'is_a', 'skill']
['commute time', 'is_a', 'job benefit']
['company culture', 'is_a', 'job benefit']
['salary', 'is_a', 'job benefit']
['Google', 'is_a', 'tech company']
['Apple', 'is_a', 'tech company']
['San Francisco', 'is_a', 'big city']
['Miami', 'is_a', 'big city']
['New York', 'is_a', 'big city']
['Software Engineer', 'is_a', 'Job role']
['Registered Nurse', 'is_a', 'Job role']
['DevOps Engineer', 'is_a', 'Job role']
['health insurance', 'is_a', 'job benefit']
['pto', 'is_a', 'job benefit']
```

¹ Кортеж в программировании – это неизменяемая упорядоченная коллекция, которая может содержать элементы различных типов. В кортеже нельзя заменить значение элемента, добавить или удалить элемент. – *Прим. ред.*

Как вы можете видеть из этого списка, сосредоточившись на извлечении фиксированного типа связи (и наиболее распространенной – связи «is a»), мы можем сгенерировать хороший, чистый список таксономических фактов с более конкретным термином (гипонимом), указывающим на более общий термин (гипероним) с ребром is_a. Рисунок 5.3 наглядно демонстрирует этот сгенерированный граф.

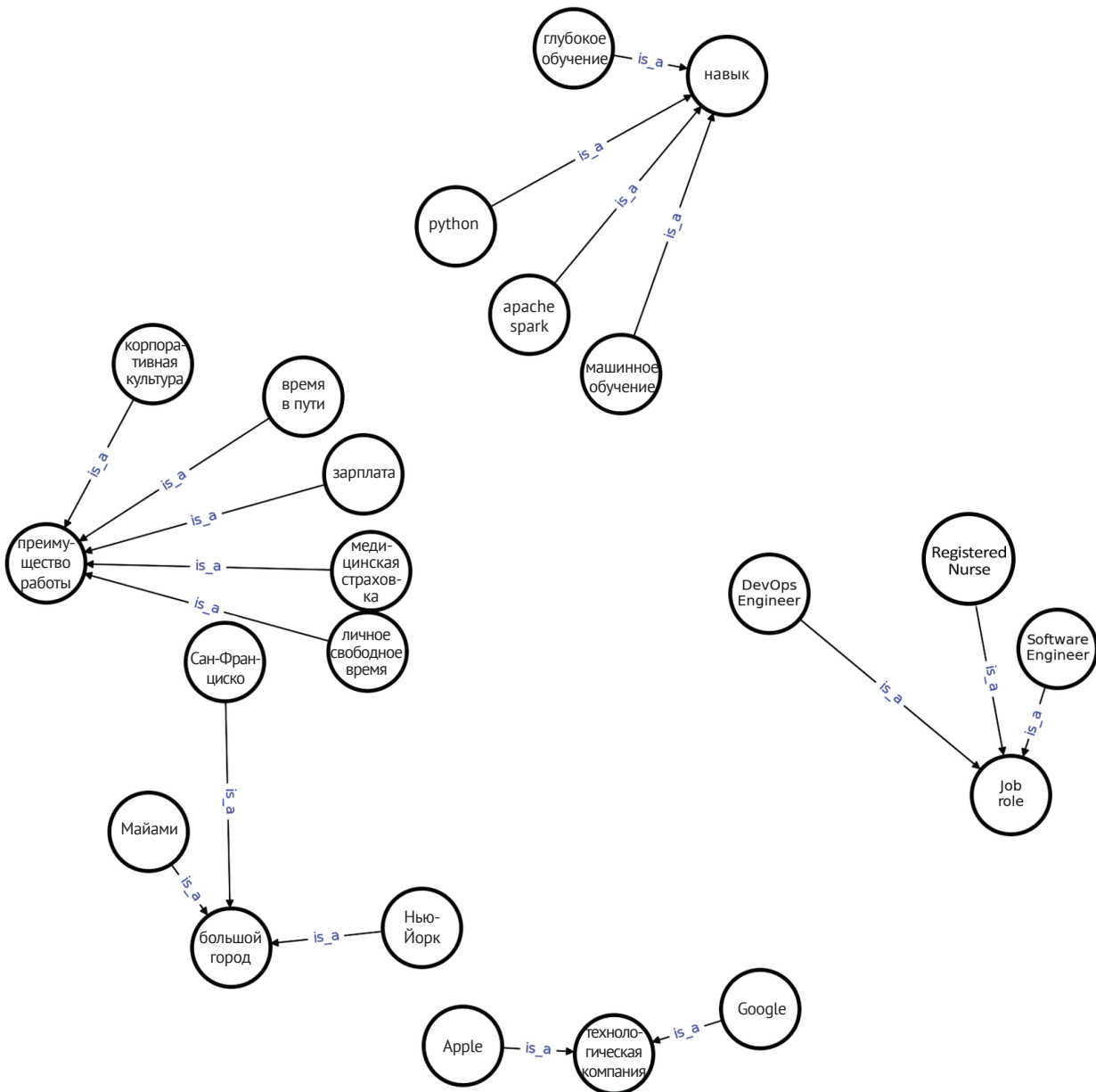


Рис. 5.3. Граф знаний, полученный на основе шаблонов Херста. Мы видим, что все узлы соединены с другими узлами через ребро «is_a»

Непоследовательность и шум, которые существуют при извлечении произвольных связей, значительно уменьшаются за счет использования шаблонов Херста. У нас все еще может быть неоднозначность в отношении связи между похожими терминами (например, орфографические ошибки, альтернативные варианты написания, известные фразы или синонимы), но их гораздо легче разрешить.

Фактически мы потратим всю следующую главу на обсуждение того, как изучить этот вид домен-специфического языка ваших сигналов и контента, чтобы использовать его при интерпретации входящих пользовательских запросов.

Хотя может быть полезно извлекать информацию из нашего текста в явный граф знаний для последующего обхода, реальность такова, что этот вид извлечения является процессом с потерями, поскольку представление элементов отключается от исходного контекста этих элементов в нашем контенте (окружающий текст и документы, содержащий текст). В следующем разделе мы представим совершенно другой вид графа знаний – семантический граф знаний, – который оптимизирован для обеспечения его обхода в реальном времени и ранжирования отношений между терминами и фразами в нашем контенте без необходимости явного построения графа и без отделения терминов от их исходного текстового контекста.

5.4. Изучение намерений путем обхода семантических графов знаний¹

В главе 2, разделы 2.1 и 2.2, мы обсудили миф о том, что текстовый контент является неструктурированными данными и что на самом деле текстовые документы представляют собой гиперструктурированные данные. Мы обсудили гипотезу распределения («слово должно быть известно компании своих соседей») и рассмотрели, что последовательности символов, термины, фразы и другие произвольные последовательности терминов можно рассматривать как нечеткие внешние ключи, связывающие схожие понятия в разных документах. Мы также обсудили, как эти связи между документами можно рассматривать как ребра в гигантском графе отношений, что позволяет нам узнавать контекстное значение терминов и сущностей, присутствующих в нашем корпусе документов.

В этом разделе мы познакомимся с семантическим графом знаний, инструментом и методом, которые позволят нам пройти по этому гигантскому графу семантических связей, присутствующих в наших документах.

5.4.1. Что такое семантический граф знаний?

Семантический граф знаний (SKG) – это «компактная, автоматически сгенерированная модель для обхода в реальном времени и ранжиро-

¹ Намерение в программировании, англ. *intent*, – это одна из центральных концепций при разработке чат-ботов. Это совокупность алгоритмов, которые система осуществляет, распознав потребность пользователя. Для обучения интенгов используются тренировочные фразы – примеры запросов. Их можно добавить в классификатор из внешних файлов с логами, а также разметить прямо из аналитики по диалогам уже после того, как бот был запущен. – *Прим. ред.*

вания любых связей в домене»¹. Мы можем представить себе SKG как поисковую систему, которая вместо сопоставления и ранжирования документов находит и ранжирует термины, которые лучше всего соответствуют запросу.

Например, если мы проиндексировали коллекцию документов по темам здравоохранения и выполнили поиск по слову *advil*, вместо возврата документов, содержащих термин для обозначения обезболивающего «*advil*», SKG автоматически (без необходимости ручного создания списка или моделирования данных) вернет значения, подобные этим:

```
advil 0.71  
motrin 0.60  
aleve 0.47  
ibuprofen 0.38  
alleve 0.37
```

Такие результаты можно рассматривать как «динамические синонимы», но вместо терминов, имеющих одинаковое значение, они больше похожи на понятийно связанные термины. Вы можете расширить лексический поисковый запрос для *advil*, включив эти другие термины, чтобы улучшить отзыв результатов поиска или повисить документы, которые понятийно соответствуют значению «*advil*», а не просто строку, содержащую пять символов *a, d, v, i, l*.

Помимо поиска связанных терминов, SKG может перемещаться между полями в вашем инвертированном индексе («найти наиболее связанные с этой должностью навыки»), перемещаться на несколько уровней в глубину («найти наиболее связанные названия должностей для этого запроса, а затем найти наиболее связанные навыки для этого запроса и каждого из этих названий должностей») и использовать любой произвольный запрос, который вы можете отправить в поисковую систему в качестве узла в обходе графа, чтобы найти семантически связанные термины в любой области.

Варианты использования SKG разнообразны. Их можно использовать для расширения запроса, генерации рекомендаций на основе контента, классификации запросов, устранения неоднозначности запросов, обнаружения аномалий, очистки данных и предиктивной аналитики. Мы рассмотрим некоторые из них в оставшейся части этой главы, но сначала давайте настроим несколько наборов данных для тестирования нашего SKG.

¹ Грейнджер и др., «Семантический граф знаний: компактная, автоматически сгенерированная модель для обхода в реальном времени и ранжирования любых связей в домене». Международная конференция IEEE по науке о данных и передовой аналитике (DSAA), стр. 420–429. ИИЭР, 2016.

5.4.2. Индексирование наборов данных

SKG лучше всего работает с наборами данных, где больше совпадений терминов, используемых вместе в документах. Чем чаще два слова встречаются в документах, тем лучше мы можем определить, встречаются ли эти термины статистически чаще, чем мы ожидали бы.

Хотя «Википедия» часто является хорошим начальным набором данных для многих вариантов использования, она обычно имеет одну страницу по основной теме, которая должна быть авторитетной, поэтому в большинстве документов нет значительного совпадения, что делает «Википедию» плохим набором данных для этого варианта использования. Напротив, большинство других веб-сайтов, на которых пользователи отправляют контент (вопросы, сообщения на форумах, объявления о вакансиях, сообщения в социальных сетях, обзоры), как правило, имеют отличные наборы данных для варианта использования SKG.

Для этой главы мы выбрали два основных набора данных: набор данных о вакансиях (объявления на доске объявлений) и ряд дампов данных Stack Exchange, включая сообщения со следующих форумов:

- здоровье;
- научная фантастика;
- взаимодействие разработчиков;
- путешествия;
- приготовление пищи.

5.4.3. Структура SKG

Чтобы наилучшим образом использовать SKG, полезно понимать, как работает граф на основе его базовой структуры.

В отличие от традиционного графа знаний, который должен быть явно смоделирован из узлов и ребер, SKG *материализуется* из базового инвертированного индекса вашей поисковой системы. Это означает, что все, что вам нужно сделать для создания SKG, – это индексировать документы в поисковой системе. Никакого дополнительного моделирования данных не требуется.

Затем инвертированный индекс и соответствующий прямой индекс служат базовой структурой данных, которая позволяет в реальном времени выполнять обход и ранжирование любых произвольных семантических отношений, присутствующих в вашей коллекции документов.

Рисунок 5.4 демонстрирует, как документы добавляются как в прямой индекс, так и в инвертированный индекс. Слева на рисунке вы можете видеть три документа, каждый из которых имеет поля `job_title`, `desc` и `skills`. Правая сторона рисунка показывает, как эти документы отображаются в вашей поисковой системе. Мы видим, что инвертированный индекс преобразует каждое поле в список терминов, а затем преобразует каждый термин в список сообщений, содержащий список документов (вместе с позициями в документах, а также некоторыми другими данными, не включенными в иллюстрацию). Это позволяет быстро и эффективно искать любой термин в любом поле и находить набор всех документов, содержащих этот термин.

Документы

id: 1
job_title: Software Engineer
desc: software engineer at a great company
skills: .Net, C#, java

id: 2
job_title: Registered Nurse
desc: a registered nurse at hospital doing hard work
skills: oncology,phlebotomy

id: 3
job_title: Java Developer
desc: a software engineer or a java engineer doing work
skills: java,scala, hibernate

Документы-термины,
прямой индекс

field	doc	term
desc	1	a
		at
		company
		engineer
		great
		software
	2	a
		at
		doing
		hard
		hospital
		nurse
		registered
		work
	3	a
		doing
		engineer
		java
		or
job_title	1	Software Engineer
...	...	...

Документы-термины,
обратный индекс

field	term	postings list	
		doc	pos
desc	a	1	4
		2	1
		3	1, 5
	at	1	3
		2	4
	company	1	6
	doing	2	6
		3	8
	engineer	1	2
		3	3, 7
	great	1	5
	hard	2	7
	hospital	2	5
	java	3	6
	nurse	2	3
	or	3	4
	registered	2	2
	software	1	1
		3	2
	work	2	10
		3	9
job_title	java developer	3	1
...	...	...	...

Рис. 5.4. Обратный индекс и прямой индекс. Документы добавляются в обратный индекс, который сопоставляет документы со списками терминов, и в прямой индекс, который сопоставляет термины обратно со списками документов. Возможность сопоставлять оба направления окажется важной для обхода графа и обнаружения связей

В дополнение к хорошо известному инвертированному индексу вы также можете увидеть менее известный *прямой индекс* в центре рис. 5.4. Прямой индекс можно рассматривать как *неинвертированный индекс*: для каждого поля он сопоставляет каждый документ со списком терминов, содержащихся в этом документе. Прямой индекс – это то, что поисковые системы используют для генерации *фасетов* (также называемых *агрегациями*) в результатах поиска, которые показывают верхние значения для каждого поля из набора документов. В поисковых системах на основе Lucene, таких как Solr, OpenSearch и Elasticsearch, прямой индекс обычно генерируется во время индексирования для поля путем включения признака, называемого *doc values*¹ для поля. В качестве альтер-

¹ Doc values в программировании – это структура данных на диске, созданная для улучшения производительности аналитических операций, таких как агрегации и сортировка. Она позволяет хранить значения в столбцовом формате, что более эффективно для сортировки и агрегации. – *Прим. ред.*

нативы Apache Solr также позволяет вам генерировать тот же прямой индекс путем «обратного инвертирования» инвертированного индекса в памяти во время запроса, что позволяет выполнять фасетирование даже для полей, для которых doc values не были добавлены в индекс.

Если у вас есть возможность искать произвольные запросы и находить наборы документов с помощью инвертированного индекса (обход от терминов к документам), а также у вас есть возможность брать произвольные наборы документов и искать термины в этих документах (обход от документов к терминам), это означает, что, выполняя два перехода (от терминов к документам и далее к терминам), вы можете найти все связанные термины, которые появляются в любых документах, соответствующих запросу. Рисунок 5.5 демонстрирует, как может происходить такой переход, включая представление структуры данных, представление теории множеств и представление графа.

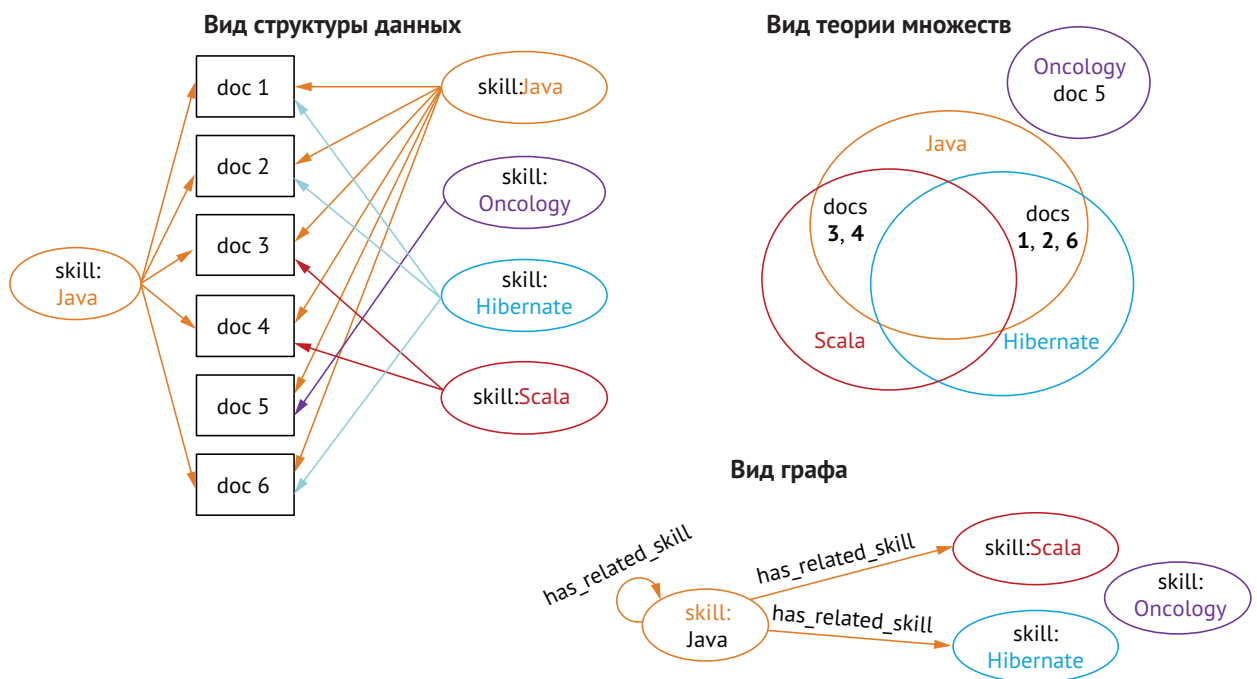


Рис. 5.5. Три представления SKG. Представление структуры данных показывает термины, сопоставленные с наборами документов, представление теории множеств показывает, как пересечение наборов документов формирует связь между ними, а представление графа показывает узлы и ребра

В представлении структуры данных, которое представляет наши инвертированные и прямые индексы, мы видим, как термины связаны с документами на основе того, появляются ли они в них. Эти связи отношений присутствуют только в том случае, если есть пересечение между документами, в которых любые два узла (термины в данном случае) появляются в представлении теории множеств. Представление графа, наконец, демонстрирует третье представление в тех же базовых структурах данных, но в этом случае мы видим узлы (вместо наборов документов) и ребра (вместо пересекающихся наборов документов). По сути, SKG существует как абстракция поверх инвертиро-

ванного индекса, который уже построен и обновляется каждый раз, когда поисковая система индексирует контент.

Обычно мы считаем, что основная функция поисковых систем – это принятие запроса, поиск соответствующих документов и возврат этих документов в порядке релевантности. Мы посвятили всю главу 3 обсуждению этого процесса, пройдя через сопоставление (разделы 3.2.4–3.2.6), ранжирование TF-IDF (раздел 3.1) и обычно используемую функцию ранжирования BM25 (раздел 3.2.1). Однако с SKG мы фокусируемся на сопоставлении и ранжировании связанных *терминов*, а не связанных документов.

Любой произвольный запрос (все, что вы можете разрешить в набор документов) может быть узлом в вашем графе, и вы можете перейти от этого узла к любому другому термину (или произвольному запросу) в любом поле документа. Кроме того, поскольку каждый обход ребра между двумя узлами использует как инвертированный индекс (термины в документы), так и прямой индекс (документы в термины), тривиально объединить эти обходы в многоуровневый обход графа, как показано на рис. 5.6.

На этом рисунке представление структуры данных показывает обход от узла навыка (Java) к слою других узлов навыков (Java, Oncology, Hibernate и Scala), к слою узлов названий должностей (Software Engineer, Data Scientist и Java Developer). Вы можете видеть, что не все узлы связаны – например, узел Oncology не отображается в представлении графа, потому что ни один из исходных узлов не может подключиться к нему через какие-либо ребра – нет перекрывающихся документов.

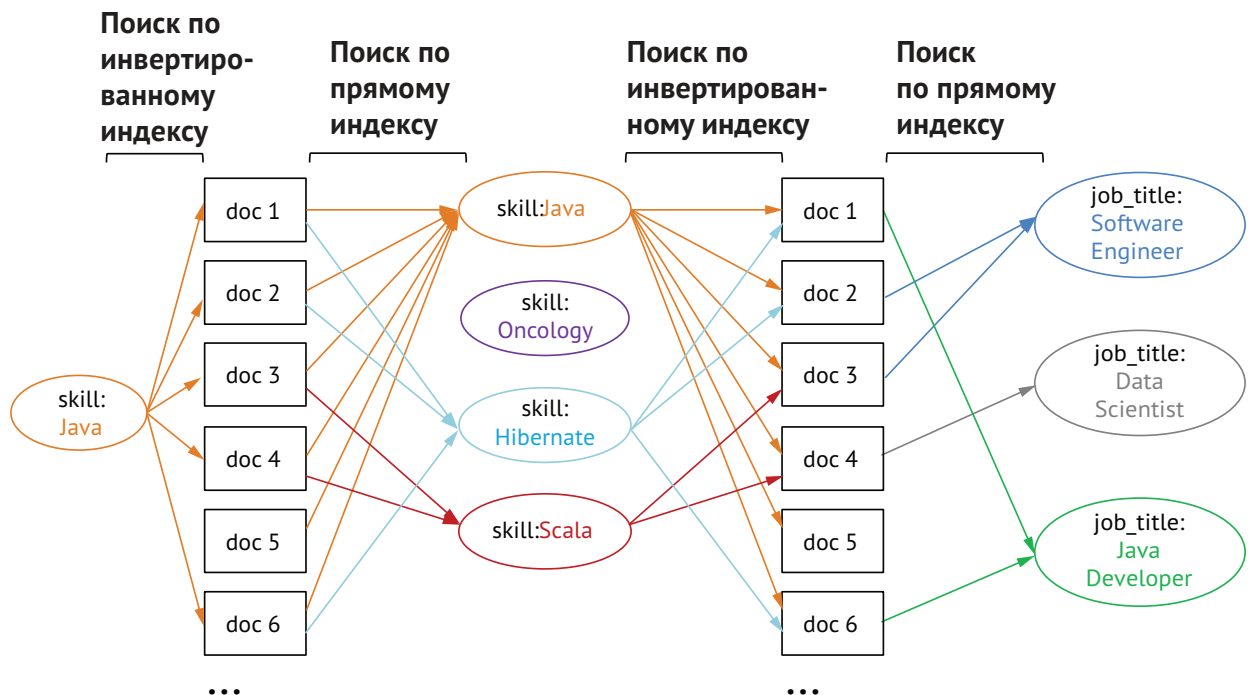
Учитывая, что не все возможные узлы будут релевантны для любого данного обхода, также важно, чтобы SKG могли оценивать и назначать вес отношениям между узлами, чтобы эти ребра могли быть приоритетными во время любого обхода графа. Мы рассмотрим оценку и назначение весов ребрам в следующем разделе.

5.4.4. Расчет весов ребер для измерения связанности узлов

Учитывая, что основная функция SKG заключается в обнаружении соответствующих семантических связей между узлами, способность вычислять *семантическое сходство* имеет решающее значение. Но что именно представляет собой семантическое сходство?

Если вы помните, гипотеза распределения, представленная в разделе 2.3, гласит, что слова, появляющиеся вместе в одних и тех же контекстах и с похожими распределениями, как правило, имеют схожие значения. Интуитивно это имеет смысл – термины «rain» или «swelling» с большей вероятностью будут встречаться в документах, в которых также упоминаются «advil», «ibuprofen» или «ice pack», чем в некоторых случайных документах. Интересно, однако, что «ice pack» может также встречаться в документах, содержащих такие термины, как «cooler», «road trip» или «cold», тогда как «advil» и «ибупрофен», скорее всего, не будут.

Вид структуры данных



Вид графа

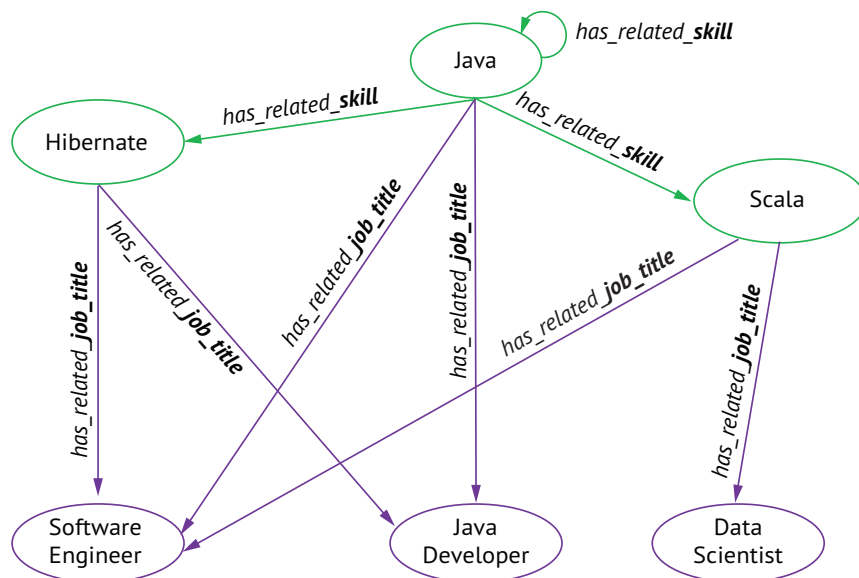


Рис. 5.6. Многоуровневый обход графа. В представлении структуры данных мы видим два обхода: через инвертированный индекс, а затем каждый раз через прямой индекс. В представлении структуры графа мы видим соответствующий двухуровневый обход: от навыков к навыкам и к названиям должностей

Эти примеры показывают слова (и их контексты) со схожими значениями, но давайте также рассмотрим такие слова, как «a», «the», «of», «and», «if», «they», и бесчисленное множество других очень распространенных стоп-слов. Эти слова также будут часто появляться в тех же кон-

текстах, что и «rain», «swelling», «advil», «ibuprofen» или любых других рассмотренных нами слов. Это указывает на вторую часть гипотезы распределения – что слова также должны встречаться с похожими распределениями. По сути, это означает, что при наличии некоторого количества документов, содержащих первый термин, любой второй термин имеет тенденцию быть семантически сопутствующим с первым термином, если он встречается в тех же документах, что и первый термин, чаще, чем в документах с другими случайными терминами.

На практике, поскольку «the» или «a» имеют тенденцию встречаться вместе почти со всеми другими терминами, они не считаются семантически схожими с этими терминами, даже если уровень их совместной встречаемости высок. Однако такие термины, как «rain» и «ibuprofen», статистически встречаются вместе гораздо чаще, чем любой из терминов появляется со случайными другими терминами, поэтому они считаются семантически схожими.

Следующее уравнение демонстрирует один из способов вычисления семантической связанности (relatedness) термина с набором документов:

$$\text{связанность}(x, fg, bg) = \frac{|D_x \cap D_{fg}| - |D_{fg}| \cdot P_x}{\sqrt{|D_{fg}| \cdot P_x \cdot (1 - P_x)}}$$

где

- x – запрос (обычно термин или последовательность терминов), для которого вычисляется связанность относительно другого запроса, приоритетного запроса fg . D_x – набор документов, соответствующих запросу x ;
- D_{fg} – это набор документов, соответствующих запросу переднего плана fg . Связанность x вычисляется относительно этого набора переднего плана;
- D_{bg} – это набор документов, соответствующих запросу заднего плана bg . Этот запрос bg должен быть некоррелирован с x и fg и обычно устанавливается для соответствия всей коллекции документов D или случайной выборке из D ;
- P_x – это вероятность нахождения x в случайном документе в наборе заднего плана, вычисляемая как $(|D_{bg} \cap D_x|) / |D_{bg}|$.

Этот расчет связанности (концептуально аналогичный z-оценке в нормальном распределении) опирается на концепцию набора документов переднего плана и набора документов заднего плана и позволяет статистически сравнивать распределение термина x между двумя наборами. Например, если набор переднего плана состоял из всех документов, соответствующих запросу «rain», а фоновый набор – из всех документов, то связанность термина «advil» будет мерой того, насколько чаще «advil» встречается в документах, также содержащих слово «rain» (набор переднего плана), по сравнению с любым случайным документом (фоновый набор). Чаще всего нормируют оценку связанности с помощью сигмоидальной функции для сопоставления

значений от -1.0 до 1.0 , где 0.0 указывает на отсутствие связи между терминами. Для простоты мы будем полагаться на этот нормированный диапазон значений в коде и всех последующих примерах.

Если два термина сильно связаны, их связанность будет положительным числом, приближающимся к 1.0 . Если термины сильно не связаны (т. е. они, как правило, встречаются только в расходящихся доменах), оценка будет ближе к -1.0 . Наконец, термины, которые вообще не связаны семантически, например стоп-слова, будут иметь оценку связанности, близкую к нулю.

Apache Solr имеет возможности SKG, встроенные непосредственно в его API фасетирования. Фасетирование обеспечивает возможность перехода от терминов к наборам документов к терминам, а функция агрегации родства (RelationnessAgg) реализует расчет семантического сходства, который мы только что описали. Следующий листинг демонстрирует поиск семантически связанных с «advil» терминов в наборе данных о здоровье Stack Exchange.

Листинг 5.4. Обнаружение терминов, семантически связанных с advil

```
health_skg = get_skg(engine.get_collection("health"))

nodes_to_traverse = [{"field": "body",
                      "values": ["advil"]},
                     {"field": "body",
                      "min_occurrences": 2,
                      "limit": 8}]

traversal = health_skg.traverse(*nodes_to_traverse)
print_graph(traversal, "advil")
```

Уменьшает шум, исключая термины, не найденные по крайней мере столько раз. →

Поле, в котором нужно найти значения для начального узла. ←

Наш начальный узел – запрос «advil». ←

Сколько узлов (терминов) будет возвращено. ←

Выполняет обход графа. ←

Выводит результаты обхода SKG. ←

Вывод:

```
advil 0.70986
motrin 0.59897
aleve 0.4662
ibuprofen 0.38264
alleve 0.36649
tylenol 0.33048
naproxen 0.31226
acetaminophen 0.17706
```

Как вы можете видеть, из всех терминов в сообщениях форума в наборе данных о здоровье Stack Exchange ранжированный порядок наиболее семантически связанных с advil терминов был списком похожих обезболивающих. Это магия использования гипотезы распределения для обнаружения и ранжирования терминов по семантическому сходству – она дает нам возможность автоматически в режиме реального времени обнаруживать связи, которые можно использовать для дальнейшего улучшения нашего понимания входящих запросов.

Ниже приведен запрос Solr SKG, который использует API фасетирования JSON Solr и возможность сортировки по функции – расчет `relatedness`, который мы только что обсудили.

```
{
  "limit": 0,
  "params": {
    "q": "*",
    "fore": "{!${defType} v=$q}",
    "back": "*",
    "defType": "edismax",
    "f0_0_query": "advil"
  },
  "facet": {
    "f0_0": {
      "type": "query",
      "query": "{!edismax qf=body v=$f0_0_query}",
      "field": "body",
      "sort": {"relatedness": "desc"},
      "facet": {"relatedness": {"type": "func",
                                "func": "relatedness($fore,$back)"},
    },
    "f1_0": {
      "type": "terms",
      "mincount": 2,
      "limit": 8,
      "sort": {"relatedness": "desc"},
      "facet": {"relatedness": {"type": "func",
                                "func": "relatedness($fore,$back)"}}
    }
  }
}
```

Функция `skg.traverse(*nodes_to_traverse)` в листинге 5.4 абстрагирует этот специфичный для движка синтаксис, но если вы пытаетесь понять нюансы того, как ваша конкретная поисковая система или векторная база данных внутренне обрабатывает эти виды обходов графа знаний, вы можете проверить функцию в блокнотах. В дальнейшем мы покажем абстракцию `skg.traverse`, но вы всегда можете вызвать функцию `skg.transform_request(*nodes_to_traverse)` напрямую, чтобы увидеть и отладить внутренний, специфичный для движка запрос.

В следующем разделе мы обсудим, как можно применять связанные термины, возвращаемые этим обходом SKG, для повышения релевантности запроса.

5.4.5. Использование SKG для расширения запроса

Сопоставление и ранжирование исключительно по ключевым словам, введенным во время поиска, не всегда обеспечивает достаточный контекст для поиска и ранжирования лучших результатов. В этих случаях вы можете значительно улучшить качество результатов поиска, динамически расширяя или иным образом дополняя запросы, чтобы включить понятийно связанные термины. В этом разделе мы рассмотрим, как можно генерировать эти связанные термины, и продемон-

стрируем несколько стратегий применения терминов для повышения качества результатов поиска.

Учитывая его способность начинать с любого ключевого слова или запроса и находить другие тесно связанные термины в любой области, одним из очевидных вариантов использования SKG является динамическое расширение запросов для включения связанных терминов. Этот тип расширения иногда называют *разреженным лексическим расширением*, поскольку оно работает с разреженными векторами токенов запросов, созданных из основанных на терминах (лексических) признаков. Одним из известных методов реализации такого типа расширения запросов является SPLADE (модель разреженной лексики и расширения), которую мы рассмотрим в разделе 7.4.3. Семантические графы знаний также предоставляют отличный способ создания контекстных, разреженных лексических расширений и имеют то преимущество, что они не требуют дополнительной тонкой настройки вашего набора данных. Это позволяет сопоставлять документы, даже если они не обязательно содержат точные ключевые слова, введенные пользователем, но они содержат другие термины, которые несут очень похожее значение. Например, вместо запроса пользователя для `advil` расширенный запрос с усиленными (подтверженными бустингу) терминами, сгенерированный SKG, может выглядеть примерно так: `advil OR motrin^0.59897 OR aleve^0.4662 OR ibuprofen^0.3824 OR ....`

Давайте рассмотрим шаги для реализации такого рода расширения запроса, используя на этот раз набор данных из другого домена (набор данных Stack Exchange scifi). Следующий листинг показывает первый шаг в этом процессе: поиск неясного термина (*obscure term*) как узла в SKG и поиск других связанных терминов (как связанных узлов в SKG). В этом случае мы будем использовать запрос для `vibranium` в качестве нашего начального узла.

Листинг 5.5. Обнаружение контекста для неизвестного термина «vibranium»

```
stackexchange_skg = get_skg(engine.get_collection("stackexchange"))

query = "vibranium"
nodes_to_traverse = [{"field": "body", "values": [query]},
                     {"field": "body", "min_occurrences": 2, "limit": 8}]

traversal = stackexchange_skg.traverse(*nodes_to_traverse)

print_graph(traversal, query)
```

Ответ:

```
vibranium 0.94237
wakandan 0.8197
adamantium 0.80724
wakanda 0.79122
```

```
alloy 0.75724
maclain 0.75623
klaw 0.75222
america's 0.74002
```

Для тех, кто не знаком с термином «вибрианиум», это прочный вымышленный металл, который существует в комиксах и фильмах Marvel (наиболее популяризированный благодаря голливудскому хиту 2018 года «Черная пантера»). Наиболее связанные термины, которые вернулись, были связаны с «Ваканданом» и «Вакандой», вымышленной культурой и страной, из которой происходит вибрианиум, «адамантием», еще одним прочным (вымышленным) металлом из комиксов Marvel, и именами «Маклейн» и «Клоу», персонажами комиксов Marvel, которые тесно связаны с металлом вибрианиумом. Маклейн создал сплав вибрианиума, из которого сделан щит Капитана Америки, отсюда и родство этих слов.

Автоматически сгенерированный граф знаний очень эффективен для определения связанных фрагментов информации. Используя SKG и расширяя свой запрос, чтобы включить дополнительный связанный контекст, вы можете радикально улучшить отзыв ваших поисковых запросов. Улучшая результаты, которые наилучшим образом соответствуют вашему запросу понятийно (а не просто тексту), вы также можете улучшить точность ваших высокоранжированных результатов поиска.

В следующем листинге показан пример перевода этого исходного запроса вместе с выводом SKG в расширенный запрос.

Листинг 5.6. Расширение запроса с узлами в SKG

```
expansion = ""
for term, stats in traversal["graph"][0]["values"][query] \
    ["traversals"][0]["values"].items():
    expansion += f'{term}^{stats["relatedness"]}'
expanded_query = f'{query}^5 ' + expansion

print(f"Expanded Query:\n{expanded_query}")
```

Расширенный запрос:

```
vibranium^5 vibranium^0.94237 wakandan^0.8197 adamantium^0.80724
wakanda^0.79122 alloy^0.75724 maclain^0.75623 klaw^0.75222
america's^0.74002
```

В этом случае мы выполняем простой поиск Boolean OR для любого из ключевых слов, связанных с исходным запросом `vibranium`, увеличивая вес исходного термина запроса в 5 раз и взвешивая влияние каждого последующего термина на оценку релевантности на основе его оценки семантической схожести. Выбор увеличения исходного термина в 5 раз произволен – вы можете выбрать здесь любое значение, чтобы назначить относительное повышение релевантности по сравнению с другими (расширенными) терминами.

Вы также можете заметить, что термин «vibranium» появляется дважды – сначала как исходный термин, а затем снова как расширенный термин (поскольку термин *также* наиболее семантически похож на себя). Это будет почти всегда иметь место, если вы ищете отдельные ключевые слова, но, поскольку ваш запрос может содержать фразы или другие конструкции, которые делают исходный запрос отличным от возвращенных терминов (если таковые имеются), обычно хорошей идеей является включение исходного запроса в качестве части расширенного (переписанного) запроса, чтобы фактический запрос пользователя всегда был представлен в результатах.

Хотя предыдущий расширенный запрос должен ранжировать результаты достаточно хорошо (приоритизируя документы, соответствующие нескольким связанным терминам), он также в значительной степени ориентирован на *recall* (расширение для включения всего релевантного) в отличие от *precision* (гарантия того, что все включенное является релевантным). Расширенный запрос может быть построен многими способами в зависимости от ваших основных целей.

Переписанные запросы могут выполнять простое расширение, требовать минимальный процент или количество терминов для соответствия, требовать определенные термины, такие как исходный запрос, для соответствия или даже просто изменять рейтинг того же исходного набора результатов. В следующем листинге показано несколько примеров, использующих минимальные пороги соответствия и проценты, которые могут сдвинуть шкалу между точностью и отзывом по мере необходимости.

Листинг 5.7. Различные стратегии расширения запроса

```
def generate_request(query, min_match=None, boost=None):
    request = {"query": query,
               "query_fields": ["title", "body"]}
    if min_match:
        request["min_match"] = min_match
    if boost:
        request["query_boosts"] = boost
    return request

simple_expansion = generate_request(f"{query} {expansion}", "1")
increased_conceptual_precision = \
    generate_request(f"{query} {expansion}", "30%")
increased_precision_same_recall = \
    generate_request(f"{query} AND ({expansion})", "2")
slightly_increased_recall = generate_request(f"{query} {expansion}",
"2")
same_results_better_ranking = generate_request(query, "2",
expansion)
```

Давайте рассмотрим окончательные поисковые запросы для каждого из предыдущих методов расширения запроса.

Простое расширение запроса: `simple_expansion`

```
{
  "query": "vibranium vibranium^0.94237 wakandan^0.8197
adamantium^0.80724
    ↳ wakanda^0.79122 alloy^0.75724 maclain^0.75623
klaw^0.75222
    ↳ america's^0.74002 ",
  "query_fields": ["title", "body"],
  "min_match": "0%"}

```

Это простое расширение запроса такое же, как описано ранее, сопоставляет любые документы, содержащие либо исходный запрос, либо любые семантически связанные термины.

Запрос повышенной точности, сниженного ответа (*increased-precision, reduced-recall query*): *increased_conceptual_precision*

```
{
  "query": "vibranium AND (vibranium^0.94237 wakandan^0.8197
    ↳ adamantium^0.80724 wakanda^0.79122 alloy^0.75724
    ↳ maclain^0.75623 klaw^0.75222 america's^0.74002)",
  "query_fields": ["title", "body"],
  "min_match": "30%"}

```

Этот пример повышенной точности, сниженного ответа указывает порог минимального соответствия в 30 %, что означает, что для соответствия документа он должен содержать не менее 30 % (округленно вниз) терминов в запросе.

Повышенная точность верхних результатов без снижения ответа: *increased_precision_same_recall*

```
{
  "query": "vibranium AND (vibranium^0.94237 wakandan^0.8197
    ↳ adamantium^0.80724 wakanda^0.79122 alloy^0.75724
    ↳ maclain^0.75623 klaw^0.75222 america's^0.74002)",
  "query_fields": ["title", "body"],
  "min_match": "2"}

```

Этот запрос повышенной точности, одинакового ответа требует соответствия термина «vibranium», и он ранжирует документы выше, когда совпадают другие термины расширения, что приводит к повышению точности верхних результатов.

Запрос с немного увеличенным ответом: *slightly_increased_recall*

```
{
  "query": "vibranium vibranium^0.94237 wakandan^0.8197
    ↳ adamantium^0.80724 wakanda^0.79122 alloy^0.75724
    ↳ maclain^0.75623 klaw^0.75222 america's^0.74002",
  "query_fields": ["title", "body"],
  "min_match": "2"}

```

Этот запрос с немного увеличенным возвратом требует соответствия двух терминов, но он явно не требует исходного запроса, поэтому его можно расширить на другие документы, которые понятийно похожи, но не обязательно содержат исходный термин запроса. По-

сколько термин «vibranium» повторяется дважды, любые документы, содержащие только «vibranium», также будут соответствовать.

Те же результаты, лучший понятийный рейтинг: `same_results_better_ranking`

```
{"query": "vibranium",  
 "query_fields": ["title", "body"],  
 "min_match": "2",  
 "query_boosts": "vibranium^0.94237 wakandan^0.8197 adamantium^0.80724  
                  ➔wakanda^0.79122 alloy^0.75724 macclain^0.75623  
                  ➔klaw^0.75222 america's^0.74002 "}
```

Этот окончательный запрос возвращает те же документы, что и исходный запрос `vibranium`, но он ранжирует их по-разному в зависимости от того, насколько хорошо они соответствуют семантически похожим терминам из графа знаний. Это гарантирует, что ключевое слово существует во всех сопоставленных документах и что все документы, содержащие запрос пользователя, будут возвращены, а также значительно улучшает рейтинг, повышая более контекстно релевантные документы.

Конечно, существует неограниченное количество возможных перестановок запросов, которые вы можете изучить при переписывании своего запроса для включения расширенного семантического контекста, но предыдущие примеры должны дать хорошее представление о типах доступных вариантов и компромиссах, которые вам следует учитывать.

5.4.6. Использование SKG для рекомендаций на основе контента

В последнем разделе мы рассмотрели, как мы можем расширить запросы, обнаружив и используя связанные узлы из SKG, включая несколько способов структурирования переписанных запросов для оптимизации точности, отзыва или даже улучшенного понятийного ранжирования по тем же результатам. Помимо расширения запросов семантически связанными терминами, также можно использовать SKG для генерации рекомендаций на основе контента путем перевода документов в запросы на основе семантического сходства терминов в документах.

Поскольку узлы в SKG могут представлять любой произвольный запрос, мы можем брать термины из документов и моделировать их как произвольные узлы для оценки относительно некоторого известного контекста документа. Это означает, что мы можем взять десятки или сотни терминов из документа, оценить их все относительно темы документа, а затем использовать наиболее семантически схожие термины для создания запроса, наилучшим образом представляющего нюансное, контекстное значение документа.

В следующем листинге представлен пример перевода документа, классифицированного как «звездные войны», и ранжирования всех терминов в документе относительно этой темы.

Листинг 5.8. Расчет того, насколько термины документа связаны со словом «star wars»

```
from aips import extract_phrases
stackexchange_skg = get_skg(engine.get_collection("stackexchange"))
classification = "star wars"
document = """this doc contains the words luke, magneto, cyclops,
            darth vader, princess leia, wolverine, apple, banana,
            galaxy, force, blaster, and chloe."""
parsed_document = extract_phrases (document)
nodes_to_traverse = [{"field": "body", "values": [classification]},
                    {"field": "body", "values": parsed_document}]
traversal = stackexchange_skg.traverse(*nodes_to_traverse)
print_graph(traversal, classification)
```

Оцененные узлы:

```
luke 0.75212
force 0.73248
darth vader 0.69378
galaxy 0.58693
princess leia 0.50491
blaster 0.47143
this 0.19193
the 0.17519
words 0.10144
and 0.09709
contains 0.03434
doc 0.00885
chloe 0.0
cyclops -0.01825
magneto -0.02175
banana -0.0319
wolverine -0.03362
apple -0.03894
```

В этих результатах вы можете увидеть список терминов из документа, который хорошо упорядочен на основе семантического сходства с темой «star wars». Термины с более низкими оценками не будут иметь никакой связи или будут иметь отрицательную связь с указанной темой. Следующий листинг фильтрует термины со связью по крайней мере выше 0.25, чтобы получить очень чистый список релевантных терминов из документа.

Листинг 5.9. Формирование запроса рекомендаций из оцененных фраз

```
def get_scored_terms(traversal):
    return {term: data["relatedness"]
            for term, data in traversal["graph"][0]["values"]["star wars"]}

\
    ["traversals"][0]["values"].items() }
```



```

rec_query = " ".join(f'"{term}"^{score}'
                      for term, score in get_scored_terms(traversal).
                      items())
                      if score > 0.25)
print(f"Expanded Query:\n{rec_query}")

```

Расширенный запрос:

```

"luke"^0.75212 "force"^0.73248 "darth vader"^0.69378 "galaxy"^0.58693
"princess leia"^0.50491 "blaster"^0.47143

```

Следующий листинг демонстрирует последний шаг в этом процессе – запуск поиска для возврата лучших документов, наиболее семантически похожих на исходный документ.

Листинг 5.10. Выполнение запроса рекомендаций на основе контента

```

stackexchange_collection = engine.get_collection("stackexchange")

request = {"query": rec_query,
           "query_fields": ["title", "body"],
           "return_fields": ["title"],
           "limit": 5,
           "filters": [("title", "*")]}

response = stackexchange_collection.search(**request)
print(json.dumps(response["docs"], indent=2))

```

Вывод:

```

[{"title": "At the end of Return of the Jedi, did Darth Vader learn
    ➔that Princess Leia was his daughter?"},
 {"title": "Did Luke know the 'Chosen One' prophecy?"},
 {"title": "Was Darth Vader at his strongest during Episode III?"},
 {"title": "Why couldn't Snoke or Kylo Ren trace Luke using the
Force?"},
 {"title": "Does Kylo Ren know that Darth Vader reconciled with Luke?"}]

```

То, что мы только что создали, – это алгоритм рекомендаций на основе контента. Когда недостаточно поведенческих сигналов пользователя для рекомендаций на основе сигнала, таких как совместная фильтрация (см. раздел 4.2.3), подход на основе контента может генерировать рекомендации, которые по-прежнему учитывают контекст и домен.

Пример в этом разделе сгенерировал базирующийся на контенте запрос рекомендаций на основе терминов, найденных в исходном документе, но стоит помнить, что SKG не ограничивается использованием переданных терминов. Вы можете добавить дополнительный уровень к обходу, чтобы найти дополнительные термины, которые семантически связаны с терминами в исходном документе, но фактически не содержатся в нем. Это может быть особенно полезно для нишевых тем,

где недостаточно документов соответствуют запросу рекомендаций – дальнейший обход откроет новые возможности для исследования.

В следующем разделе мы сделаем быстрый шаг за пределы графовых отношений «связано с» и посмотрим, сможем ли мы использовать SKG для генерации и обхода некоторых более интересных ребер.

5.4.7. Использование SKG для моделирования произвольных отношений

До сих пор все наши обходы SKG использовали отношение «связано с» («is related to»). То есть мы находили силу семантической связи между двумя словами или фразами с помощью функции `relatedness`, но мы только измеряли, что узлы «связаны», а не то, *как* они связаны. Что, если бы мы могли найти другие виды ребер между узлами вместо просто ребер типа «связано с»?

Если вы помните, узлы в SKG материализуются на лету путем выполнения запроса, который соответствует набору документов. Если узел, с которого вы начинаете, – `engineer`, этот узел внутренне представлен как набор всех документов, содержащих слово «инженер». Если узел помечен как `software engineer`, этот узел внутренне представлен как набор всех документов, содержащих термин «программное обеспечение», пересекающихся со всеми документами, содержащими термин «инженер». Если поиск выполняется по запросу `"software engineer" OR java`, то он внутренне представлен как объединение набора всех документов, содержащих термин «software» на одну позицию перед термином «engineer» (фраза), с набором всех документов, содержащих термин «java». Все запросы, независимо от их сложности, внутренне представлены как набор документов.

Вы также можете вспомнить, что ребро формируется путем нахождения набора документов, содержащих оба узла. Это означает, что и *узлы*, и *ребра* внутренне представлены с использованием одного и того же механизма – набора документов. Практически это означает, что, если мы можем построить узел с помощью запроса, который аппроксимирует интересную связь (в отличие от сущности), мы можем связать два узла вместе через «узел связи» (`relationship node`) аналогично тому, как ребро использовалось бы для связи узлов вместе в традиционной структуре графа.

Давайте рассмотрим пример. Возвращаясь к нашему набору данных научной фантастики, предположим, что мы хотим задать вопрос о Джин Грей, одном из популярных персонажей франшизы Marvel Comics X-Men. В частности, предположим, что мы хотим выяснить, кто был влюблен в Джин Грей.

Мы можем сделать это, используя начальный узел `jean grey`, переходя к узлу `in love with`, а затем запросив топовые термины, связанные с `in love with` в контексте `jean grey`. Листинг 5.11 демонстрирует этот запрос. Проходя через узел, предназначенный для захвата явного лингвистического отношения (в данном случае `in love with`), мы можем использовать промежуточный узел для построения ребра между начальным и конечным узлами.

Листинг 5.11. Материализация ребра через «узел соотношения»

```
scifi_skg = get_skg(engine.get_collection("scifi"))
starting_node = "jean grey"
relationship = "in love with"
nodes_to_traverse = [{"field": "body", "values": [starting_node]},
                     {"field": "body", "values": [relationship],
                      "default_operator": "OR"},
                     {"field": "body",
                      "min_occurrences": 25, "limit": 10}]

traversal = scifi_skg.traverse(*nodes_to_traverse)
print_graph(traversal, starting_node, relationship)
```

Вывод:

```
jean 0.84915
grey 0.74742
summers 0.61021
cyclops 0.60693
xavier 0.53004
wolverine 0.48053
mutant 0.46532
x 0.45028
mutants 0.42568
magneto 0.42197
```

Если вы не знакомы с этими персонажами, вот соответствующая предыстория Джин Грей: у нее повторяющиеся отношения с двумя мутантами – одним по имени Циклоп (настоящее имя: Скотт Саммерс) и одним по имени Росомаха. Кроме того, и это неизвестно большинству фанатов, двое наставников Джин Грей, профессор Чарльз Ксавье и Магнето, были известны тем, что имели любовный интерес к Джин Грей в разное время на протяжении серии комиксов.

Если мы изучим результаты из листинга 5.11, в списке мы увидим все эти ожидаемые имена. Первые два термина, «джин» и «грей», являются наиболее связанными, так как мы ищем влюбленность относительно Джин Грей. Ее имя будет семантически очень связано с самим собой. Следующие два термина, «саммерс» и «циклоп», оба относятся к одному и тому же человеку, самому выдающемуся любовному интересу Джин. Затем мы видим «ксавье» и «росомаха», а последний результат в списке – «магнето». На рис. 5.7 показаны некоторые из основных графических взаимосвязей для этого обхода.

Используя промежуточный узел (т. е. in love with) для моделирования отношений между другими узлами, мы можем сформировать любое произвольно типизированное ребро между узлами, если только мы можем выразить это ребро как поисковый запрос.

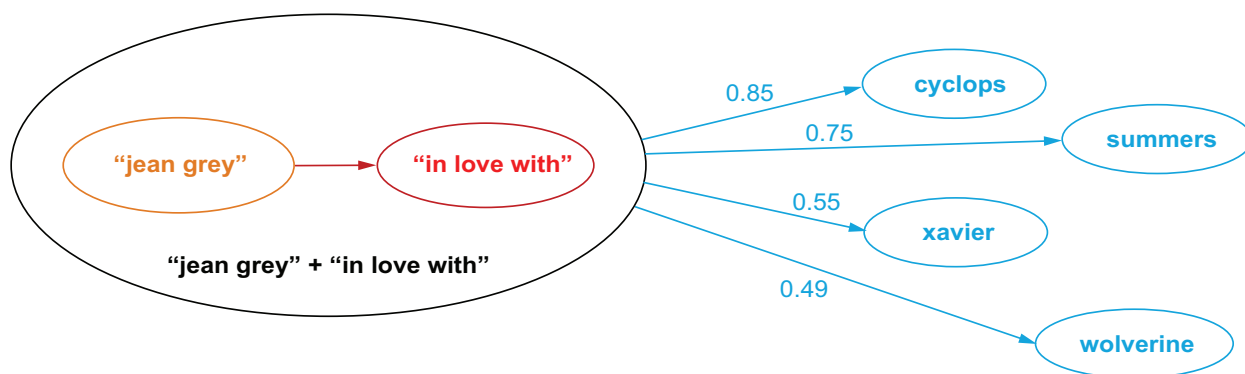


Рис. 5.7. Прохождение произвольно определенных типов ребер. Материализуя новый узел с объединенным контекстом как исходного узла («jean grey»), так и нового узла («in love with»), мы можем пройти от этого объединенного узла («jean grey» + «in love with») к другим узлам. Это эквивалентно тому, что мы проходим от «jean grey» через ребро «in love with» к другим узлам

Хотя результаты нашего обхода графа в листинге 5.11 были довольно хорошими, мы видим, что термины «х» (предположительно из «x-men») и «mutant» также появляются. Джин Грей и все другие перечисленные люди являются мутантами в комиксах X-Men, поэтому эти термины так семантически связаны. Однако эти термины не являются хорошими ответами на вопрос «Кто влюблен в Джин Грей?».

Это поднимает важный момент: SKG – это статистический граф знаний. Существование отношения in love with основано исключительно на статистических корреляциях терминов в нашей коллекции, поэтому, как и в любом подходе онтологического обучения, будет шум. Тем не менее для автоматически сгенерированного графа без явного моделирования сущностей эти результаты довольно хороши.

Если бы мы хотели улучшить качество этих результатов, одним из самых простых действий было бы запустить предварительную обработку контента для идентификации сущностей (людей, мест и вещей) и индексировать их вместо просто отдельных ключевых слов. Это привело бы к тому, что вместо отдельных ключевых слов («лето», «ксавье», «джин», «грей») были бы возвращены реальные имена людей (например, «Скотт Саммерс», «Чарльз Ксавье», «Джин Грей»).

Также стоит отметить, что обход отношений полностью зависит от того, обсуждались ли эти отношения в базовом корпусе документов. В этом случае во многих сообщениях на форуме обсуждаются отношения каждого из этих людей с Джин Грей. Если бы существовало недостаточно документов, возвращенные результаты могли бы быть плохими или отсутствовать. Чтобы избежать шума в наших результатах, мы установили пороговое значение min_occurrences равным 25, указывающее, что должно существовать не менее 25 документов, в которых обсуждаются jean grey, in love with и другие найденные и оцененные узлы. Мы рекомендуем установить min_occurrences на некоторое число больше 1, чтобы избежать ложных срабатываний.

Хотя обход произвольных лингвистических отношений, таких как «влюблен в», может быть полезен с исследовательской точки зрения, с точки зрения понимания запроса обычно достаточно придержи-

ваться отношения по умолчанию «связан с» и использовать оценки родства между терминами для большинства случаев использования семантического поиска. Однако все еще может быть полезно проходить через несколько уровней отношений, чтобы создать лучший контекст. В частности, может быть полезно переходить от термина к полю классификации, чтобы предоставить некоторый дополнительный контекст, а затем к связанным значениям термина в этой категории. Мы рассмотрим эту стратегию более подробно в главе 6, где сосредоточимся на устранении неоднозначности терминов с несколькими значениями.

5.5. Использование графов знаний для семантического поиска

Предоставляя возможность принимать произвольные запросы и динамически обнаруживать связанные термины с учетом контекста, SKG становятся ключевым инструментом для интерпретации запросов и ранжирования релевантности. Мы увидели, что SKG не только могут помочь интерпретировать и расширять запросы, но и могут предоставлять возможности для классификации запросов и ключевых слов в режиме реального времени и устранения неоднозначности множественных значений терминов в каждом запросе.

В начале главы мы также исследовали, как строить явные графы знаний с помощью открытых методов извлечения информации. Что может быть пока неочевидно – так это то, как анализировать произвольные входящие запросы и искать соответствующий контекст и сущности в графе знаний. Мы потратим большую часть главы 7 на то, как построить сквозную семантическую поисковую систему, которая может анализировать запросы и интегрировать эти возможности графа знаний.

Есть еще несколько критических видов отношений, которые нам нужно добавить в наш граф знаний и которые важны для поисковых систем, такие как орфографические ошибки, синонимы и домен-специфичные фразы. Мы рассмотрим, как автоматически изучать каждый из этих источников домен-специфичной терминологии из пользовательских сигналов или контента, в следующей главе, которая фокусируется на изучении домен-специфичного языка.

Резюме

- Графы знаний моделируют отношения между сущностями в вашей области (домене) и могут быть построены явно с известными отношениями или могут быть извлечены динамически из вашего контента.
- Извлечение открытой информации, процесс извлечения фактов из вашего контента (субъект, отношение, триплеты объектов) может использоваться для изучения произвольных отношений

(что обычно приводит к зашумленным данным) или для извлечения отношений гипонима и гиперонима (менее зашумленных) из текста в явный граф знаний.

- Семантические графы знаний (SKG) позволяют обходить и ранжировать произвольные семантические отношения между любым контентом в вашем поисковом индексе. Это позволяет вам использовать ваш индексированный контент напрямую как граф знаний и языковую модель без необходимости дополнительного моделирования данных.
- Рекомендации на основе контента, которые не полагаются на сигналы пользователя, могут быть сгенерированы путем ранжирования наиболее семантически интересных терминов и фраз.



Использование контекста для изучения домен-специфического языка

В этой главе рассматривается:

- классификация намерения запроса;
- разрешение неоднозначности смысла запроса;
- определение ключевой терминологии из пользовательских сигналов;
- изучение связанных фраз из пользовательских сигналов;
- изучение опечаток и альтернативных вариантов терминов из пользовательских сигналов.

В главе 5 мы продемонстрировали, как генерировать и использовать семантический граф знаний (SKG), а также как явно извлекать сущности, факты и отношения в граф знаний. Оба метода основаны на навигации либо по лингвистическим связям между терминами в одном документе, либо по статистической совместной встречаемости терминов в нескольких документах и контекстах. Вы научились ис-

пользовать графы знаний для поиска связанных терминов и как эти связанные термины могут интегрироваться в различные стратегии переписывания запросов для повышения отзыва или точности.

В этой главе мы глубже погрузимся в понимание намерения запроса и нюансы использования различных контекстов для интерпретации терминологии, специфичной для предметной области, в запросах. Мы начнем с изучения классификации запросов, а затем покажем, как эти классификации можно использовать для устранения неоднозначности запросов с несколькими потенциальными значениями. Оба подхода расширят наше использование SKG из предыдущей главы.

Хотя эти подходы на основе SKG более эффективны для контекстуализации и интерпретации запросов, они по-прежнему полагаются на наличие высококачественных документов, которые точно представляют ваш домен. В результате их эффективность для интерпретации пользовательских запросов зависит от того, насколько хорошо запросы пересекаются с искомым контентом.

Например, если 75 % ваших пользователей ищут одежду, но большую часть вашего инвентаря составляют фильмы и цифровые медиа, то когда они ищут запрос *shorts* (шорты и т. п.) и все результаты представляют собой видео с коротким временем показа (известными как *digital shorts*, в смысле цифровые короткие видео), большинство ваших пользователей будут сбиты с толку результатами. Учитывая данные в ваших журналах запросов, было бы лучше, если бы «shorts» можно было сопоставить с другими связанными терминами, которые чаще встречаются в ваших сигналах запроса, такими как «pants» (брюки), «clothing» (одежда) и «shirts». Может быть очень полезно не только полагаться на содержание ваших документов, чтобы узнать связи между терминами и фразами, но и использовать ваши пользовательские сигналы. В этой главе мы продемонстрируем методы извлечения ключевых фраз, изучения связанных фраз и выявления распространенных ошибок или альтернативных написаний на основе пользовательских сигналов. Используя как контекст на основе контента, так и поведенческий контекст из реальных взаимодействий пользователя, ваша поисковая система будет лучше понимать домен-специфичную терминологию и фактическое намерение пользователя.

6.1. Классификация намерения запроса

Цель или намерение запроса обычно важнее ключевых слов. Поиск по слову *driver crashed* может означать две совершенно разные вещи в контексте новостей или туристического контента по сравнению с контекстом компьютерных технологий¹. Аналогично кто-то, ищущий в электронной коммерции определенное название продукта или его идентификатор, вероятно, ищет очень конкретный продукт с высо-

¹ Driver может означать как водителя автомобиля, так и компьютерную программу, управляющую неким компьютерным устройством. – Прим ред.

кой вероятностью желая его купить. Общий поиск, например `kitchen appliances`, может указывать на то, что пользователь просто намерен просмотреть доступные продукты, чтобы узнать, что есть в наличии.

В обоих контекстах классификатор запросов может быть эффективным при определении общего вида выдаваемого запроса. В зависимости от домена контекст запроса может быть применен автоматически (например, фильтрация категории документов), использован для изменения алгоритма релевантности (автоматическое повышение определенных продуктов) или даже использован для управления другим пользовательским опытом (пропуск страницы результатов и переход непосредственно на страницу определенного продукта). В этом разделе мы покажем, как использовать SKG из главы 5 в качестве классификатора для входящих запросов для построения классификатора запросов.

Обход SKG выполняет поиск k -ближайших соседей на каждом уровне обхода графа. K -ближайший сосед – это тип классификации, который берет точку данных (например, запрос или термин) и пытается найти k самых похожих других точек данных в векторном пространстве. Если у нас есть поле, такое как `category` или `classification` в наших документах, мы можем попросить SKG «найти категорию с наивысшей степенью родства с моим начальным узлом». Поскольку начальным узлом обычно является запрос пользователя, SKG может классифицировать этот запрос.

Мы продолжим использовать индексированные наборы данных Stack Exchange в качестве SKG, который будет расширен для классификации запросов (в этом разделе) и устранения неоднозначности в смысле запроса (в разделе 6.2).

В листинге 6.1 показана функция, которая принимает запрос пользователя и проходит по SKG, чтобы найти семантически связанные категории для классификации запроса. Поскольку мы проиндексировали несколько различных категорий Stack Exchange (научная фантастика, здоровье, кулинария, `devops`¹ и т. д.), мы будем использовать эти категории в качестве наших классификаций.

Листинг 6.1. Классификация запросов с использованием SKG

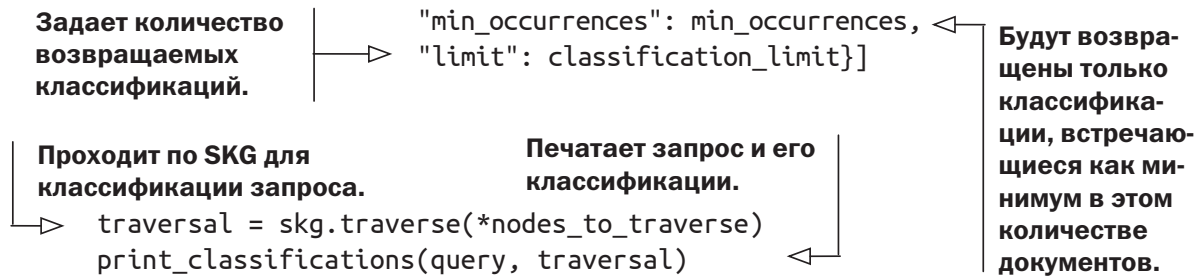
Начальный узел графа на основе запроса, сопоставленного с полем.

Поле, из которого мы найдем связанные классификации. В этом случае мы переходим к полю категории.

```
def print_query_classification(query, classification_field="category",
                             classification_limit=5, keywords_field="body", min_occurrences=5):
```

```
    nodes_to_traverse = [{"field": keywords_field,
                          "values": [query]},
                        {"field": classification_field,
```

¹ DevOps в программировании – это методология непрерывной разработки программного обеспечения, которую используют программисты, тестировщики и системные администраторы. – Прим. ред.



```

skg = get_skg(get_engine().get_collection("stackexchange"))

print_query_classification("docker", classification_limit=3)
print_query_classification("airplane", classification_limit=1)
print_query_classification("airplane AND crash", classification_limit=2)
print_query_classification("vitamins", classification_limit=2)
print_query_classification("alien", classification_limit=1)
print_query_classification("passport", classification_limit=1)
print_query_classification("driver", classification_limit=2)
print_query_classification("driver AND taxi", classification_limit=2)
print_query_classification("driver AND install", classification_limit=2)

```

Примеры классификаций запросов:

Query: docker
Classifications:
devops 0.87978

Query: airplane
Classifications:
travel 0.33334

Query: airplane AND crash
Classifications:
scifi 0.02149
travel 0.00475

Query: vitamins
Classifications:
health 0.48681
cooking 0.09441

Query: alien
Classifications:
scifi 0.62541

Query: passport
Classifications:
travel 0.82883

Query: driver
Classifications:
travel 0.38996
devops 0.08917

```
Query: driver AND taxi
Classifications:
  travel 0.24184
  scifi -0.13757
```

```
Query: driver AND install
Classifications:
  devops 0.22277
  travel -0.00675
```

Этот запрос использует SKG для поиска k ближайших соседей на основе сравнения семантического сходства между запросом и каждой доступной классификацией (в поле category).

Мы видим баллы классификации для каждой потенциальной категории для каждого запроса, при этом `airplane` и `passport` классифицируются как «путешествие», `vitamins` классифицируются как «здоровье» и «кулинария», а `alien` классифицируется как «научная фантастика». Однако, когда мы уточняем запрос `airplane` до более конкретного запроса, например `airplane AND crash`, мы видим, что классификация меняется с «путешествие» на «научная фантастика», потому что документы об авиакатастрофах с большей вероятностью встречаются в документах «научная фантастика», чем в документах «путешествие».

В качестве другого примера водитель может иметь несколько значений. Он возвращает две потенциальные классификации («путешествие» или «devops»), при этом категория «путешествие» является очевидным выбором, когда не указан никакой другой контекст. Однако, когда предоставляется дополнительный контекст, мы видим, что запрос `driver AND taxi` соответствующим образом классифицируется в категории «путешествия», а `driver AND install` соответствующим образом классифицируется в категории «devops».

Эта способность SKG находить семантические связи между произвольными комбинациями терминов делает его полезным для оперативной классификации входящих запросов. Вы можете автоматически применять классификации как фильтры запросов или бустинга, направлять запросы на контекстно-зависимый алгоритм или целевую страницу или автоматически устранять неоднозначность терминов запроса. В следующем разделе мы рассмотрим использование двухуровневого обхода графа для реализации устранения неоднозначности смысла запроса.

6.2. Устранение неоднозначности смысла запроса

При интерпретации намерений пользователей из их запросов сложно точно понять, что они имели в виду под каждым словом. Проблема полисемии, или неоднозначности терминов, может существенно повлиять на результаты поиска.

Если кто-то ищет `server`, это может относиться к тому, кто принимает заказы и обслуживает столики в ресторане, или это может означать компьютер, который запускает программное обеспечение в сети. В идеале мы хотим, чтобы наша поисковая система могла устранять неоднозначность каждого из этих значений слова и генерировать уникальный список связанных терминов в каждом разрешенном контексте. Рисунок 6.1 демонстрирует эти два потенциальных контекста для слова «server» и виды связанных терминов, которые можно найти в каждом контексте.

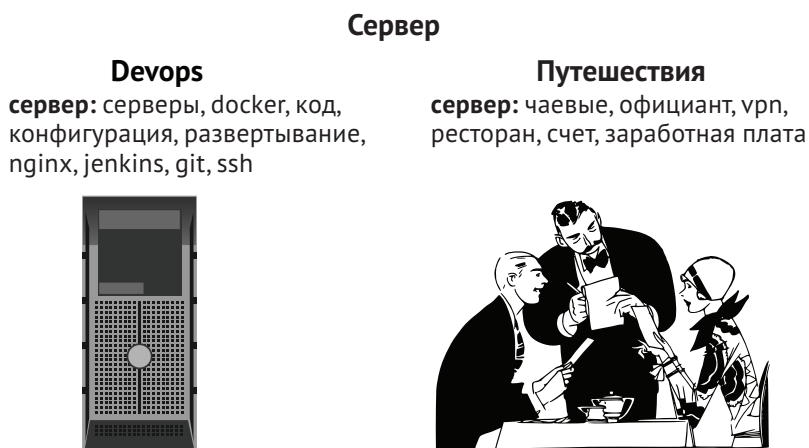


Рис. 6.1. Различение множественных значений неоднозначного термина «сервер»

В разделе 6.1 мы продемонстрировали, как использовать SKG для автоматической классификации запросов в набор известных категорий. Учитывая, что мы уже знаем, как классифицировать наши запросы, добавление обхода второго уровня может предоставить контекстуализированный список связанных терминов для каждой классификации запроса.

Другими словами, переходя от запроса к классификации, а затем к терминам, мы можем сгенерировать список терминов, которые описывают контекстуализированную интерпретацию исходного запроса в каждой из верхних классификаций. Следующий листинг показывает функцию, которая устраняет неоднозначность запроса таким образом, используя SKG.

Листинг 6.2. Устранение неоднозначности намерения запроса в разных контекстах

```
def print_query_disambiguation(query,
    context_field="category", context_limit=5,
    keywords_field="body", keywords_limit=10, min_occurrences=5):
```

```
    nodes_to_traverse = [{"field": keywords_field, | Начальный узел обхода графа  
                        "values": [query]}, | (запрос пользователя).
```

Первый обход возвращает контексты для устранения неоднозначности запроса.

Второй обход – из ключевых слов, связанных как с запросом, так и с каждым связанным контекстом.

```
    {"field": context_field,
     "min_occurrences": min_occurrences,
     "limit": context_limit},
    {"field": keywords_field,
     "min_occurrences": min_occurrences,
     "limit": keywords_limit}]
```



```
traversal = skg.traverse(*nodes_to_traverse)
print_disambigutations(query, traversal)
```

Из этого листинга видно, что поле контекста (поле `category` по умолчанию) и поле ключевых слов (поле `body` по умолчанию) используются как часть двухуровневого обхода. Для любого переданного запроса мы сначала находим наиболее семантически связанную категорию, а затем термины, наиболее семантически связанные с исходным запросом в этой категории.

Следующий листинг демонстрирует, как вызвать эту функцию, передавая три разных запроса, содержащих неоднозначные термины, для которых мы хотим найти дифференцированные значения.

Листинг 6.3. Запуск устранения неоднозначности в смысле запроса для нескольких запросов

```
print_query_disambigutaion("server")
print_query_disambigutaion("driver", context_limit=2)
print_query_disambigutaion("chef", context_limit=2)
```

Результаты запросов в листинге 6.3 можно найти в табл. 6.1–6.3, за которыми следует поисковый запрос SKG, используемый для устранения неоднозначности `chef` в листинге 6.4. Каждый контекст устранения неоднозначности (поле `category`) оценивается относительно запроса, а каждое обнаруженное ключевое слово (`body`) оценивается относительно как запроса, так и контекста устранения неоднозначности.

Таблица 6.1. Списки связанных терминов, контекстуализированные по категориям для запроса `server`

Query: server	
Context: devops 0.83796 Keywords: server 0.93698 servers 0.76818 docker 0.75955 code 0.72832 configuration 0.70686 deploy 0.70634 nginx 0.70366 jenkins 0.69934 git 0.68932 ssh 0.6836	Context: cooking -0.1574 Keywords: server 0.66363 restaurant 0.16482 pie 0.12882 served 0.12098 restaurants 0.11679 knife 0.10788 pieces 0.10135 serve 0.08934 staff 0.0886 dish 0.08553
Context: travel -0.15959 Keywords: server 0.81226 tipping 0.54391 vpn 0.45352 tip 0.41117	Context: scifi -0.28208 Keywords: server 0.78173 flynn's 0.53341 computer 0.28075 computers 0.2593

Таблица 6.1. Списки связанных терминов, контекстуализированные по категориям для запроса `server` (продолжение)

Query: server	
servers 0.39053	flynn 0.24963
firewall 0.33092	servers 0.24778
restaurant 0.21698	grid 0.23889
tips 0.19524	networking 0.2178
bill 0.18951	shutdown 0.21121
cash 0.18485	hacker 0.19444

Таблица 6.1 показывает верхние наиболее семантически связанные категории для запроса `server`, за которыми следуют наиболее семантически связанные ключевые слова из поля `body` в каждом из этих контекстов категорий. На основе данных мы видим, что категория «devops» является наиболее семантически связанной (положительная оценка 0.83796), тогда как следующие три категории содержат отрицательные оценки (-0.1574 для «cooking», -0.15959 для «travel» и -0.28208 для «scifi»). Таким образом, для сервера запросов категория «devops» является подавляюще наиболее вероятной категорией, которая будет релевантной. Если мы посмотрим на различные списки терминов, которые возвращаются для каждой из категорий, мы также увидим несколько различных значений. В категории «devops» значение термина «server» сосредоточено на инструментах, связанных с управлением, сборкой и запуском кода на компьютерном сервере. В категории «научная фантастика» значение вращается вокруг взлома компьютерных сетей и отключения их сетей. С другой стороны, в категории «путешествия» подавляющее значение слова «сервер» связано с кем-то, кто работает в ресторане, с такими терминами, как «чаевые», «ресторан» и «счет».

При реализации интеллектуального поискового приложения с использованием этих данных, если вы знаете, что контекст пользователя связан с путешествием, имеет смысл использовать конкретное значение в категории «путешествия». Если контекст неизвестен, лучшим выбором обычно является либо наиболее семантически связанная категория, либо самая популярная категория среди ваших пользователей.

Таблица 6.2 демонстрирует разрешение неоднозначности в смысле запроса для запроса `driver`. В этом случае есть две связанные категории, причем «travel» является наиболее семантически связанной (0.38996) по сравнению с «devops» (0.08917). Мы можем видеть два очень разных значения «driver» в каждом из этих контекстов, причем «driver» в категории «travel» связан с «taxi», «car», «license», «riving» и «bus», тогда как в категории «devops» «driver» связан с «ipam», «aufs» и «overlayfs», которые являются различными видами драйверов, связанных с компьютером.

Таблица 6.2. Списки контекстуально связанных терминов по категориям для запроса driver

Query: driver	
Context: travel 0.38996	Context: devops 0.08917
Keywords:	Keywords:
driver 0.93417	ipam 0.78219
drivers 0.76932	driver 0.77583
taxi 0.71977	aufs 0.73758
car 0.65572	overlayfs 0.73758
license 0.61319	container_name 0.73483
driving 0.60849	overlay2 0.69079
taxis 0.57708	cgroup 0.68438
traffic 0.52823	docker 0.67529
bus 0.52306	compose.yml 0.65012
driver's 0.51043	compose 0.55631

Если кто-то ищет driver, он обычно не хочет получать документы об обоих значениях слова в результатах поиска. Существует несколько способов работы с несколькими потенциальными значениями запрашиваемых ключевых слов, например группировка результатов по значению для выделения различий, выбор только наиболее вероятного значения, тщательное вкрапление различных значений в результаты поиска для обеспечения разнообразия или предоставление альтернативных предложений запроса для различных контекстов. Намеренный выбор здесь обычно намного лучше, чем механическое объединение нескольких различных значений.

Таблица 6.3. Списки контекстуализированных связанных терминов по категориям для запроса chef

Query: chef	
Context: cooking 0.37731	Context: devops 0.34959
Keywords:	Keywords:
chef 0.93239	chef 0.87653
chefs 0.5151	puppet 0.79142
www.pamperedchef.com 0.41292	docs.chef.io 0.7865
kitchen 0.39127	ansible 0.73888
restaurant 0.38975	www.chef.io 0.72073
cooking 0.38332	learn.chef.io 0.71902
chef's 0.37392	default.rb 0.70194
professional 0.36688	configuration 0.68296
nakiri 0.36599	inspec 0.65237
pampered 0.34736	cookbooks 0.61503

В качестве последнего примера табл. 6.3 демонстрирует устранение неоднозначности запроса для запроса chef. Оба верхних контекста показывают достаточно положительные оценки связанности, что указывает на то, что оба значения являются вероятными интерпретациями. Хотя контекст «cooking» имеет немного более высокую оценку (0.37731), чем контекст «devops» (0.34959), все равно было бы важно


```

"f2_0": {
  "type": "terms",
  "field": "body",
  "mincount": 5, "limit": 10,
  "sort": {"relatedness": "desc"},
  "facet": {"relatedness": {"type": "func",
                             "func":
"relatedness($fore,$back)"}}}
}}}}}}

```

Окончательный обход SKG находит термины из поля body, связанные с контекстом устранения неоднозначности.

Это внутренний запрос Solr SKG, используемый для устранения неоднозначности query chef с context_limit в 2. Запрос будет специфичным для любой настроенной поисковой системы или векторной базы данных, или он будет возвращаться к Solr, если у системы нет возможностей SKG. Инструкции по изменению настроенной поисковой системы см. в приложении В.

Объединяя классификацию запросов, устранение неоднозначности терминов и расширение запроса, SKG может обеспечить расширенные возможности семантического поиска, домен-специфичные и высококонтекстуализированные возможности семантического поиска в вашей поисковой системе на базе ИИ. Мы подробнее рассмотрим использование этих методов в главе 7, когда применим их в приложении семантического поиска в реальном времени.

6.3. Изучение связанных фраз из сигналов запроса

До сих пор вы видели, как использовать свой контент в качестве графа знаний для обнаружения связанных терминов, классификации запросов и устранения неоднозначности терминов. Хотя эти методы являются эффективными, они также полностью зависят от качества ваших документов. В оставшейся части этой главы мы рассмотрим другой основной источник знаний о вашем домене – пользовательские сигналы (запросы, клики и последующие действия). Часто пользовательские сигналы могут приводить к аналогичным, если не более полезным, идеям, чем содержимое документа для интерпретации запросов.

В качестве отправной точки для изучения домен-специфичной терминологии из реального поведения пользователя давайте рассмотрим, что представляют ваши журналы запросов. Для каждого запроса к вашей поисковой системе журнал запросов содержит идентификатор человека, выполняющего поиск, запрос, который был выполнен, и временную метку запроса. Это означает, что если один пользователь ищет несколько терминов, вы можете сгруппировать эти поиски вместе, а также узнать, в каком порядке были введены термины.

Хотя это не всегда верно, одно разумное предположение заключается в том, что если кто-то вводит два разных запроса в течение очень короткого промежутка времени, второй запрос, скорее всего, будет либо уточнением первого запроса, либо запросом по связанной теме. На рис. 6.2 показана реалистичная последовательность поисков, которые вы можете найти для одного пользователя в ваших журналах запросов.



Рис. 6.2. Типичная последовательность поисков из журналов запросов для конкретного пользователя

Глядя на эти запросы, мы интуитивно понимаем, что `iphond` – это неправильное написание `iphone`, что `iphone accesories` – неправильное написание `iphone accessories` и что `iphone, pink phone case` и `pink iphone case` – все это связанные запросы. Мы разберемся с неправильным написанием в следующем разделе, но сейчас мы можем считать их также связанными терминами.

Хотя неразумно полагаться на сигналы одного пользователя, чтобы сделать вывод о том, что два запроса связаны, схожие шаблоны запросов у многих пользователей указывают на вероятные связи. Как мы продемонстрировали в разделе 5.4.5, запросы можно расширить, включив связанные термины, чтобы улучшить запоминание. В этом разделе мы рассмотрим методы изучения связанных запросов, сначала с помощью журналов запросов, а затем с помощью перекрестных ссылок на журналы взаимодействия продуктов.

6.3.1. Просмотр журналов запросов для поиска связанных запросов

Прежде чем начать поиск сигналов пользователей для поиска связанных запросов, давайте сначала преобразуем наши сигналы в более простой формат для обработки. Следующий листинг обеспечивает преобразование нашей общей структуры сигнала в простую структуру, которая сопоставляет каждое вхождение термина запроса с пользователем, искавшим этот термин.

Листинг 6.5. Сопоставление сигналов с парами
ключевое слово – пользователь

```

signals_collection = engine.get_collection("signals")
create_view_from_collection(signals_collection, "signals")
query = """SELECT LOWER(searches.target) AS keyword, searches.user
          FROM signals AS searches
          WHERE searches.type='query'"""
spark.sql(query).createOrReplaceTempView("user_searches")
print_keyword_user_pairs()

```

Загружает все документы из корпуса
сигналов в представление Spark.

Выбирает ключевые слова
и данные пользователя из
сигналов запроса.

Вывод:

Number of keyword user pairs: 725459

Keyword user pairs derived from signals:

User "u10" searched for "joy stick"

User "u10" searched for "xbox"

User "u10" searched for "xbox360"

Из этого списка вы можете видеть, что представлено более 725 000 запросов. Наша цель – найти пары связанных запросов на основе того, сколько пользователей ввели оба запроса. Чем чаще два запроса встречаются одновременно в журналах запросов разных пользователей, тем более связанными считаются эти запросы.

В следующем списке показана каждая пара запросов, где оба запроса искал один и тот же пользователь, а также количество пользователей, которые искали оба запроса (users_cooc).

Листинг 6.6. Общее количество появлений
и совместных появлений запросов

```

query = """SELECT k1.keyword AS keyword1, k2.keyword AS keyword2,
COUNT(DISTINCT k1.user) users_cooc
FROM user_searches k1
JOIN user_searches k2 ON k1.user = k2.user
WHERE k1.keyword > k2.keyword
GROUP BY k1.keyword, k2.keyword"""

spark.sql(query).createOrReplaceTempView("keywords_users_cooc")
query = """SELECT keyword, COUNT(DISTINCT user) users_occ FROM
          user_searches GROUP BY keyword"""
spark.sql(query).createOrReplaceTempView("keywords_users_oc")
print_keyword_cooccurrences()

```

Ограничивает
пары ключевых
слов только одной
перестановкой,
чтобы избежать
дублирующих пар.

Подсчитывает
количество
пользователей,
которые искали
как k1, так и k2.

Объединяет представление
user_searches с самим со-
бой в поле пользователя,
чтобы найти все пары клю-
чевых слов, которые искал
один и тот же пользователь.

Вывод:

keyword	users_occ
lcd tv	8449
ipad	7749
hp touchpad	7144
iphone 4s	4642
touchpad	4019
laptop	3625
laptops	3435
beats	3282
ipod	3164
ipod touch	2992

Number of co-occurring keyword searches: 244876

keyword1	keyword2	users_cooc
green lantern	captain america	23
iphone 4s	iphone	21
laptop	hp laptop	20
thor	captain america	18
bose	beats	17
iphone 4s	iphone 4	17
skullcandy	beats	17
laptops	laptop	16
macbook	mac	16
thor	green lantern	16

В листинге 6.6 первый запрос, как видно из результатов, выдает наиболее часто искомые ключевые слова. Хотя это могут быть самые популярные запросы, они не обязательно являются запросами, которые чаще всего встречаются вместе с другими запросами. Второй запрос выдает общее количество пар запросов (244 876), где оба запроса искали одним и тем же пользователем хотя бы один раз. Окончательный запрос ранжирует эти пары запросов по популярности. Эти верхние пары запросов тесно связаны.

Однако обратите внимание, что верхний результат имеет только 23 совместно встречающихся пользователя, что означает, что количество точек данных разрежено и, вероятно, будет включать больше шума далее по списку. В следующем разделе мы рассмотрим метод объединения сигналов по другой оси (взаимодействия продуктов), который может помочь с этой проблемой разреженности.

Хотя прямое агрегирование количества поисков в совпадающие появления пользователями помогает найти самые популярные пары запросов, популярность поисков не единственная метрика, полезная для поиска связности. Ключевые слова «and» и «of» часто встречаются вместе, как и «phones», «movies», «computers» и «electronics», поскольку все они являются общими словами, которые ищут многие люди. Чтобы дополнительно сосредоточиться на силе связи между терминами неза-

висимо от их индивидуальной популярности, мы можем использовать технику, называемую *точечной взаимной информацией*.

Точечная взаимная информация (PMI) – это мера ассоциации между любыми двумя событиями. В контексте обработки естественного языка PMI предсказывает вероятность того, что два слова встречаются вместе, потому что они связаны, по сравнению с вероятностью того, что они встречаются вместе случайно. Для расчета и нормирования PMI можно использовать множество формул, но мы будем использовать вариант, называемый PMI^k , где $k = 2$, который лучше, чем PMI, поддерживает согласованность оценок независимо от частотности слов.

Формула для расчета PMI^2 показана на рис. 6.3.

$$PMI^2(k1, k2) = \log \frac{P(k1, k2)^2}{P(k1) \cdot P(k2)}$$

Рис. 6.3. Оценка PMI2

В нашей реализации $k1$ и $k2$ представляют два разных ключевых слова, которые мы хотим сравнить. Параметр $P(k1, k2)$ показывает, как часто один и тот же пользователь ищет оба ключевых слова, тогда как $P(k1)$ и $P(k2)$ показывают, как часто пользователь ищет только первое ключевое слово или второе ключевое слово соответственно. Интуитивно понятно, что если ключевые слова появляются вместе чаще, чем можно было бы ожидать, исходя из вероятности их случайного появления вместе, то они будут иметь более высокий балл PMI^2 . Чем выше балл, тем более вероятно, что термины семантически связаны.

В следующем листинге показано вычисление PMI^2 для нашего набора данных пар совместно встречающихся запросов.

Листинг 6.7. Вычисление PMI^2 для запросов пользователей

```
query = """
SELECT k1.keyword AS k1, k2.keyword AS k2, k1_k2.users_cooc,
k1.users_occ AS n_users1, k2.users_occ AS n_users2,
LOG(POW(k1_k2.users_cooc, 2) /
    (k1.users_occ * k2.users_occ)) AS pmi2
FROM keywords_users_cooc AS k1_k2
JOIN keywords_users_oc AS k1 ON k1_k2.keyword1 = k1.keyword
JOIN keywords_users_oc AS k2 ON k1_k2.keyword2 = k2.keyword"""
spark.sql(query).createOrReplaceTempView("user_related_keywords_pmi")

spark.sql("""SELECT k1, k2, users_cooc, n_users1,
                n_users2, ROUND(pmi2, 3) AS pmi2
                FROM user_related_keywords_pmi
                WHERE users_cooc > 5 ORDER BY pmi2 DESC, k1 ASC""").
show(10)
```

Расчет индекса деловой активности (PMI).

Вывод:

k1	k2	users_cooc	n_users1	n_users2	pmi2
iphone 4s cases	iphone 4 cases	10	158	740	-7.064
sony laptops	hp laptops	8	209	432	-7.252
otterbox iphone 4	otterbox	7	122	787	-7.58
green lantern	captain america	23	963	1091	-7.594
kenwood	alpine	13	584	717	-7.815
sony laptop	dell laptop	10	620	451	-7.936
wireless mouse	godfather	6	407	248	-7.939
hp laptops	dell laptops	6	432	269	-8.08
mp3 players	dvd recorder	6	334	365	-8.128
quicken	portable dvd players	6	281	434	-8.128

Результаты из листинга 6.7 сортируются по баллу PMI², и мы устанавливаем минимальный порог вхождений >5, чтобы помочь удалить шум. Запросы «hp laptops», «hp laptops» и «sony laptops» преобразуются как связанные, а также такие бренды, как «kenwood» и «alpine». Примечательно, что в парах также есть шум, например «wireless mouse» с «godfather» и «quicken» с «portable dvd players». Одно предостережение относительно использования PMI заключается в том, что небольшое количество вхождений вместе у нескольких пользователей может привести к шуму легче, чем при использовании совместного появления, которое основано на предположении о том, что термины часто встречаются вместе.

Один из способов объединить преимущества как модели совместного появления, так и моделей PMI² – это создать составной балл. Это обеспечит сочетание популярности и вероятности появления, что должно переместить пары запросов, которые совпадают по обоим баллам, в верхнюю часть списка. Листинг 6.8 демонстрирует один из способов объединения этих двух мер. В частности, мы берем ранжированный список всех оценок совместной встречаемости (r1) вместе с ранжированным списком всех оценок PMI² (r2) и объединяем их вместе для создания составной оценки ранжирования, как показано на рис. 6.4.

$$\text{comp_score}(q1, q2) = \frac{\left(\frac{r1(q1, q2) + r2(q1, q2)}{r1(q1, q2) \cdot r2(q1, q2)} \right)}{2}$$

Рис. 6.4. Составная оценка ранжирования, объединяющая совместную встречаемость и рейтинг PMI²

Оценка `Comp_score`, или составная оценка ранжирования, показанная на рис. 6.4, присваивает высокий балл парам запросов (запрос q1 и запрос q2), где их ранг в списке совместной встречаемости (r1) и их ранг в списке PMI² (r2) высоки, и присваивает более низкий ранг по мере того, как термины перемещаются ниже в списках ранжирования. Результатом является смешанный рейтинг, который учитывает как популярность (совместную встречаемость), так и вероятность

связанности запросов независимо от их популярности (PMI^2). В следующем листинге показано, как рассчитать `comp_score` на основе уже рассчитанных оценок совместного появления и PMI^2 .

Листинг 6.8. Расчет составной оценки из совместного появления и PMI

```

query = """
SELECT *, (r1 + r2 / (r1 * r2)) / 2 AS comp_score
FROM (
    SELECT *,
    RANK() OVER (PARTITION BY 1
                  ORDER BY users_cooc DESC) r1,
    RANK() OVER (PARTITION BY 1
                  ORDER BY pmi2 DESC) r2
    FROM user_related_keywords_pmi)"""
spark.sql(query).createOrReplaceTempView("users_related_keywords_comp_score")

spark.sql("""SELECT k1, k2, users_cooc, ROUND(pmi2, 3) as pmi2,
              r1, r2, ROUND(comp_score, 3) as comp_score
              FROM users_related_keywords_comp_score
              ORDER BY comp_score ASC, pmi2 ASC""").show(20)

```

Расчет составной оценки объединяет отсортированные ранги оценки PMI^2 и сопутствующих событий.

Ранжирует оценки сопутствующих событий от лучших (самая высокая сопутствующая встречаемость) до худших (самая низкая сопутствующая встречаемость).

Ранжирует оценки PMI^2 от лучших (самый высокий PMI^2) до худших (самый низкий балл PMI^2).

Вывод:

k1	k2	users_cooc	pmi2	r1	r2	comp_score
green lantern	captain america	23	-7.594	1	8626	1.0
iphone 4s	iphone	21	-10.217	2	56156	1.25
laptop	hp laptop	20	-9.133	3	20383	1.667
thor	captain america	18	-8.483	4	13190	2.125
iphone 4s	iphone 4	17	-10.076	5	51964	2.6
bose	beats	17	-10.074	5	51916	2.6
skullcandy	beats	17	-9.001	5	18792	2.6
laptops	laptop	16	-10.792	8	80240	4.063
macbook	mac	16	-9.891	8	45464	4.063
thor	green lantern	16	-8.594	8	14074	4.063
headphones	beats by dre	15	-9.989	11	49046	5.545
macbook pro	macbook	15	-9.737	11	39448	5.545
macbook air	macbook	15	-9.443	11	26943	5.545
ipod touch	ipad	13	-11.829	14	200871	7.036
ipad 2	ipad	13	-11.765	14	196829	7.036
nook	kindle	13	-9.662	14	36232	7.036
macbook pro	macbook air	13	-9.207	14	21301	7.036
kenwood	alpine	13	-7.815	14	9502	7.036
beats by dre	beats	12	-10.814	19	82811	9.526
macbook	apple	12	-10.466	19	62087	9.526

В целом составной рейтинговый балл делает разумную работу по смешиванию наших метрик совместного появления и PMI^2 для преодоления ограничений каждого из них. Все главные результаты, показанные в листинге 6.8, выглядят разумными. Однако одна проблема, которую мы уже отметили в этом разделе, заключается в том, что числа совместного появления очень редки. В частности, самая высокая совместная встречаемость любых пар запросов из более чем 700 000 сигналов запросов составила 23 перекрывающихся пользователя для «green lantern» и «captain america», как показано в листинге 6.6.

В следующем разделе мы покажем, как можно преодолеть эту проблему с разреженными данными, когда отсутствует перекрытие между пользователями для определенных пар запросов. Мы достигнем этого, объединив множество пользователей в большую группу со схожим поведением. В частности, мы переключим наше внимание на продукты, где пользовательские запросы пересекаются, а не на отдельных пользователей, отправляющих перекрывающиеся запросы.

6.3.2. Поиск связанных запросов через взаимодействие продуктов

Метод, используемый для поиска связанных терминов в разделе 6.3.1, зависит от многих пользователей, ищущих перекрывающиеся запросы. Как мы видели, при более чем 700 000 сигналов запросов наибольшее перекрытие любой пары запросов составило 23 пользователя. Поскольку данные могут быть настолько разреженными, часто имеет смысл агрегировать их по чему-то другому, а не по пользователям.

В этом разделе мы покажем, как можно использовать тот же метод (используя совместное появление и PMI^2), но делать объединение на основе сигналов кликов по продуктам вместо пользователей. Поскольку, как мы надеемся, у вас будет гораздо больше пользователей, чем продуктов, и поскольку определенные продукты, скорее всего, будут кликнуты в ответ на похожие ключевые слова, этот метод помогает преодолеть проблему разреженности данных и создает более высокие перекрытия между запросами.

Преобразование в листинге 6.9 объединяет отдельные сигналы запросов и кликов в отдельные строки с тремя ключевыми столбцами: keyword, user и product.

Листинг 6.9. Сопоставление необработанных сигналов с группами ключевых слов, пользователей и продуктов

Использует сигналы кликов для создания групп ключевых слов, пользователей и продуктов.	<pre>query = """SELECT LOWER(searches.target) AS keyword, searches.user AS user, clicks.target AS product FROM signals AS searches RIGHT JOIN signals AS clicks ON searches.query_id = clicks.query_id WHERE searches.type = 'query' AND clicks.type = 'click'""" spark.sql(query).createOrReplaceTempView("keyword_click_product") print_signals_format()</pre>
--	---

Вывод:

Original signals format:

```
+-----+-----+-----+-----+-----+-----+
|          id|  query_id|  signal_time|  target| type|  user|
+-----+-----+-----+-----+-----+-----+
|000001e9-2e5a-4a...|u112607_0_1|2020-04-18 16:33|      amp|query|u112607|
|00001666-1748-47...|u396779_0_1|2019-10-16 10:22|Audio stand|query|u396779|
|000029d2-197d-4a...|u466396_0_1|2020-05-07 11:39|alarm clock|query|u466396|
+-----+-----+-----+-----+-----+-----+
```

Simplified signals format:

```
+-----+-----+-----+
| keyword|user|  product|
+-----+-----+-----+
|  joy stick| u10|097855018120|
|      xbox| u10|885370235876|
|virgin mobile|u100|799366521679|
+-----+-----+-----+
```

Используя эти данные, мы теперь сможем определить силу взаимосвязи между любыми двумя ключевыми словами на основе их использования среди независимых пользователей, ищущих одни и те же продукты. Листинг 6.10 генерирует пары ключевых слов для определения их потенциальной взаимосвязи для всех пар ключевых слов, где оба ключевых слова использовались в запросе для одного и того же документа. Идея поиска перекрывающихся запросов для каждого пользователя в разделе 6.3.1 заключалась в том, что каждый пользователь, скорее всего, будет искать связанные предметы. Однако каждый продукт, скорее всего, также будет искаться связанными запросами, поэтому мы можем изменить нашу ментальную модель с «найти, сколько пользователей искали оба запроса» на «найти, сколько документов было найдено обоими запросами среди всех пользователей». Результат этого преобразования в листинге 6.10 теперь включает следующие столбцы:

- $k1$, $k2$ – два ключевых слова, которые потенциально связаны, потому что оба привели к клику по одному и тому же продукту;
- n_users1 – количество пользователей, что искали $k1$ и кликнули на продукт, который также был кликнут после поиска некоторого пользователя по $k2$;
- n_users2 – количество пользователей, что искали $k2$ и кликнули на продукт, который также был кликнут после поиска некоторого пользователя по $k1$;
- $users_cooc$ – представляет общее количество пользователей, которые искали $k1$ или $k2$ и посетили продукт, посещенный другими пользователями, которые также искали $k1$ или $k2$. Рассчитывается как $n_users1 + n_users2$;
- $n_products$ – количество продуктов, на которые кликнули пользователи, искавшие как $k1$, так и $k2$.

Листинг 6.10. Пары ключевых слов, приводящие к клику по одному и тому же продукту

```

query = """
SELECT k1.keyword AS k1, k2.keyword AS k2, SUM(p1) n_users1, sum(p2)
n_users2,
SUM(p1 + p2) AS users_cooc, COUNT(1) n_products FROM (
  SELECT keyword, product, COUNT(1) AS p1 FROM keyword_click_product
  GROUP BY keyword, product) AS k1 JOIN (
  SELECT keyword, product, COUNT(1) AS p2 FROM keyword_click_product
  GROUP BY keyword, product) AS k2 ON k1.product = k2.product
WHERE k1.keyword > k2.keyword GROUP BY k1.keyword, k2.keyword"""
spark.sql(query).createOrReplaceTempView("keyword_click_product_cooc")
print_keyword_pair_data()

```

Вывод:

Number of co-occurring queries: 1579710

k1	k2	n_users1	n_users2	users_cooc	n_products
laptops	laptop	3251	3345	6596	187
tablets	tablet	1510	1629	3139	155
tablet	ipad	1468	7067	8535	146
tablets	ipad	1359	7048	8407	132
cameras	camera	637	688	1325	116
ipad	apple	6706	1129	7835	111
iphone 4	iphone	1313	1754	3067	108
headphones	head phones	1829	492	2321	106
ipad 2	ipad	2736	6738	9474	98
computers	computer	536	392	928	98
iphone 4 cases	iphone 4 case	648	810	1458	95
netbook	laptop	1017	2887	3904	94
laptop	computers	2794	349	3143	94
netbook	laptops	1018	2781	3799	91
headphones	headphone	1617	367	1984	90
laptop	hp	2078	749	2827	89
tablet	computers	1124	449	1573	89
laptops	computers	2734	331	3065	88
mac	apple	1668	1218	2886	88
tablet pc	tablet	296	1408	1704	87

Расчеты `users_cooc` и `n_products` – это два разных способа взглянуть на общее качество сигнала, чтобы понять, насколько мы уверены, что любые два термина `k1` и `k2` связаны. Результаты в настоящее время сортируются по `n_products`, и вы можете видеть, что верхняя часть списка связей довольно чистая. Эти пары ключевых слов представляют собой несколько видов значимых семантических связей, включая следующие:

- варианты написания – «laptops» ⇒ «laptop»; «headphones» ⇒ «head phones» и т. д.;

- ассоциации с брендом – «tablet» ⇒ «ipad»; «laptop» ⇒ «hp»; «mac» ⇒ «apple» и т. д.;
- синонимы/альтернативные названия – «netbook» ⇒ «laptop»; «tablet pc» ⇒ «tablet»;
- расширение категории – «ipad» ⇒ «tablet»; «iphone 4» ⇒ «iphone»; «tablet» ⇒ «computers»; «laptops» ⇒ «computers».

Вы можете написать собственные домен-специфичные алгоритмы для определения некоторых из этих конкретных типов отношений, как мы сделаем для вариантов написания в разделе 6.5.

Также можно использовать `n_users1` и `n_users2`, чтобы определить, какой из двух запросов более популярен. В случае вариантов написания мы видим, что `headphones` используются чаще, чем `head phones` (1829 против 492 пользователей), а также более распространены, чем `headphone` (1617 против 367 пользователей). Аналогичным образом мы видим, что `tablet` гораздо более распространен в использовании, чем `tablet pc` (1408 против 296 пользователей).

Хотя наш текущий список пар ключевых слов выглядит чистым, он представляет только пары ключевых слов, которые оба встречались вместе в поисковых запросах, приведших к одним и тем же продуктам. Определение популярности каждого ключевого слова в целом даст лучшее представление о том, какие конкретные ключевые слова наиболее важны для нашего графа знаний. Следующий листинг вычисляет самые популярные ключевые слова из наших сигналов запроса, которые привели по крайней мере к одному клику по продукту.

Листинг 6.11. Вычисление поисковых запросов по ключевым словам, которые привели к кликам

```
query = """SELECT keyword, COUNT(1) AS n_users FROM keyword_click_
product
GROUP BY keyword"""
spark.sql(query).createOrReplaceTempView("keyword_click_product_oc")
print_keyword_popularity()
```

Вывод:

Keyword searches that resulted in clicks: 13744

```
+-----+-----+
| keyword      | n_users |
+-----+-----+
| ipad         | 7554    |
| hp touchpad  | 4829    |
| lcd tv       | 4606    |
| iphone 4s    | 4585    |
| laptop       | 3554    |
| beats        | 3498    |
| laptops      | 3369    |
| ipod         | 2949    |
| ipod touch   | 2931    |
```

	ipad 2		2842	
	kindle		2833	
	touchpad		2785	
	star wars		2564	
	iphone		2430	
	beats by dre		2328	
	macbook		2313	
	headphones		2270	
	bose		2071	
	ps3		2041	
	mac		1851	

+-----+

Этот список идентичен списку из листинга 6.6, но вместо отображения количества пользователей, искавших ключевое слово, этот список показывает количество пользователей, которые искали ключевое слово и также кликали на продукт. Мы будем использовать это в качестве нашего основного списка запросов для расчета PMI².

Поскольку наши пары запросов и популярность запросов теперь основаны на запросах и взаимодействиях с продуктами, остальные наши вычисления (PMI² и составной балл) такие же, как в разделе 6.3.1, поэтому мы опустим их здесь (они включены в блокноты для запуска). После расчета PMI² и составных баллов следующий листинг показывает окончательные результаты наших расчетов связанных терминов на основе взаимодействия с продуктами.

Листинг 6.12. Оценка связанных терминов на основе взаимодействия с продуктами

```
query = """SELECT k1, k2, n_users1, n_users2, ROUND(pmi2, 3) AS pmi2,
              ROUND(comp_score, 3) AS comp_score
FROM product_related_keywords_comp_score
ORDER BY comp_score ASC"""
dataframe = spark.sql(query)
print("Number of co-occurring queries:", dataframe.count(), "\n")
dataframe.show(20)
```

Вывод:

Number of co-occurring queries: 1579710

k1	k2	n_users1	n_users2	pmi2	comp_score
ipad	hp touchpad	7554	4829	1.232	1.0
ipad 2	ipad	2842	7554	1.431	1.25
tablet	ipad	1818	7554	1.669	1.667
touchpad	ipad	2785	7554	1.223	2.125
tablets	ipad	1627	7554	1.749	2.6
ipad2	ipad	1254	7554	1.903	3.083
ipad	apple	7554	1814	1.5	3.571
touchpad	hp touchpad	2785	4829	1.394	4.063

ipad	hp tablet	7554	1421	1.594	4.556
ipod touch	ipad	2931	7554	0.863	5.05
ipad	i pad	7554	612	2.415	5.545
kindle	ipad	2833	7554	0.828	6.042
laptop	ipad	3554	7554	0.593	6.538
ipad	apple ipad	7554	326	2.916	7.036
ipad 2	hp touchpad	2842	4829	1.181	7.533
laptops	laptop	3369	3554	1.29	8.031
ipad	hp	7554	1125	1.534	8.529
ipads	ipad	254	7554	3.015	9.028
ipad	htc flyer	7554	1834	1.016	9.526
ipad	i pad 2	7554	204	3.18	10.025

Результаты листингов 6.11 и 6.12 показывают преимущество агрегации на менее детальном уровне. Рассматривая все запросы, которые привели к клику по определенному продукту, список пар запросов теперь намного больше, чем в разделе 6.3.1, где пары запросов были агрегированы по отдельным пользователям. Вы можете видеть, что теперь рассматривается 1 579 710 пар запросов по сравнению с 244 876 (по листингу 6.6) при агрегации по пользователю.

Кроме того, вы можете видеть, что связанные запросы включают более детальные вариации для популярных запросов (ipad, ipad 2, ipad2, i pad, ipads, i pad 2). Наличие более детальных вариаций, таких как эта, будет полезно, если вы объединяете это обнаружение связанных терминов с другими алгоритмами, такими как обнаружение орфографических ошибок, которые мы рассмотрим в разделе 6.5.

Между подходом SKG в предыдущей главе и анализом журнала запросов в этой главе вы теперь увидели несколько методов обнаружения связанных фраз. Однако прежде чем мы сможем использовать связанные фразы, нам сначала нужно уметь идентифицировать такие фразы во входящих запросах. В следующем разделе мы рассмотрим, как мы можем сгенерировать список известных фраз из наших сигналов запроса.

6.4. Обнаружение фраз из пользовательских сигналов

В разделе 5.3 мы обсудили несколько методов извлечения произвольных фраз и связей из документов. Хотя это может иметь большое значение для обнаружения всех соответствующих домен-специфичных фраз в вашем контенте, этот подход страдает от двух проблем разной природы:

- он генерирует много шума – не каждая именная группа¹ в вашем потенциально огромном наборе документов важна, и вероят-

¹ Именная группа (англ. *noun phrase*), «субстантивная группа» или «группа существительного», – лингвистический термин, определяющий группу имен в форме словосочетаний, составляющих компонент иерархической структуры предложения, которые обладают синтаксическими свойствами существительного и в которых имя существительное является вершиной такой синтаксической группы, т. е. главным словом, определяющим характеристику всей ее составляющей. – Прим. ред.

ность определения неправильных фраз (ложных срабатываний) увеличивается по мере увеличения количества ваших документов;

- он игнорирует то, что волнует ваших пользователей, – реальная мера интереса пользователей передается тем, что они ищут. Они могут быть заинтересованы только в подмножестве вашего контента или могут искать вещи, которые даже не представлены в нем должным образом.

В этом разделе мы сосредоточимся на том, как определить важные домен-специфичные фразы из ваших пользовательских сигналов.

6.4.1. Обработка запросов как сущностей

Самый простой способ извлечь сущности из журналов запросов – это обрабатывать весь запрос как одну сущность. В таких случаях использования, как наш сайт электронной коммерции RetroTech, это работает очень хорошо, так как многие из запросов представляют собой названия продуктов, категории, названия брендов, названия компаний или имена людей (актеров, музыкантов и т. д.). Учитывая этот контекст, большинство популярных запросов в конечном итоге являются сущностями, которые можно использовать напрямую как фразы без необходимости какого-либо специального анализа.

Оглядываясь на вывод листинга 6.11, вы обнаружите следующие самые популярные запросы:

keyword	n_users
ipad	7554
hp touchpad	4829
lcd tv	4606
iphone 4s	4585
laptop	3554
...	...

Это те сущности, которые принадлежат к списку известных сущностей, многие из которых являются многословными фразами. В этом случае самый простой метод извлечения сущностей также является самым мощным – просто используйте запросы в качестве списка сущностей. Чем выше частота каждого запроса среди пользователей, тем более уверенно вы можете добавлять его в список сущностей.

Один из способов уменьшить потенциальные ложные срабатывания от шумных запросов – найти фразы, которые пересекаются как в ваших документах, так и в запросах. Кроме того, если в ваших документах есть разные поля, например название продукта или компании, вы можете перекрестно ссылаться на свои запросы с этими полями, чтобы назначить тип сущностям, найденным в ваших запросах. В зависимости от сложности ваших запросов, использование наиболее распространенных поисков в качестве ключевых сущностей мо-

жет быть самым простым способом получения высококачественного списка сущностей.

6.4.2. Извлечение сущностей из более сложных запросов

В некоторых случаях запросы могут содержать больше шума (логическую структуру, расширенные операторы запросов и т. д.) и, следовательно, не могут напрямую использоваться как сущности. В этих случаях наилучшим подходом к извлечению сущностей может быть повторное применение стратегий извлечения сущностей из главы 5, но на основе сигналов вашего запроса.

Готовая лексическая поисковая система анализирует запросы как отдельные ключевые слова и ищет их в инвертированном индексе. Например, запрос для `new york city` будет автоматически интерпретироваться как логический запрос `new AND york AND city` (или, если вы установите оператор по умолчанию на `OR`, то `new OR york OR city`). Затем алгоритмы ранжирования релевантности будут оценивать каждое ключевое слово по отдельности, вместо того чтобы понимать, что определенные слова объединяются, чтобы создать фразы, которые затем приобретают другое значение.

Возможность идентифицировать и извлекать домен-специфичные фразы из запросов может обеспечить более точную интерпретацию запроса и релевантность. Мы уже продемонстрировали один способ извлечения домен-специфичных фраз из документов в разделе 5.3, используя библиотеку `sraCy NLP` для выполнения анализа зависимостей и извлечения именных групп. Хотя запросы часто слишком короткие для выполнения настоящего анализа зависимостей, все еще возможно применить некоторую фильтрацию частей речи к любым обнаруженным фразам в запросах, чтобы исключить неименные группы. Если вам нужно разделить разделы запросов, вы также можете токенизировать запросы и удалить синтаксис запросов (`AND`, `OR` и т. д.) перед поиском фраз для извлечения. Обработка определенных шаблонов запросов для вашего приложения может потребовать некоторой домен-специфичной логики анализа запросов, но если ваши запросы в основном состоят из отдельных фраз или легко токенизируются в несколько фраз, ваши запросы, вероятно, представляют собой лучший источник домен-специфичных фраз для извлечения и добавления в ваш граф знаний. Мы рассмотрим примеры кода идентификации фраз при анализе запросов в разделе 7.4.

6.5. Ошибки и альтернативные представления

Мы рассмотрели обнаружение домен-специфичных фраз и поиск связанных фраз, но есть две очень важные подкатегории связанных фраз, которые обычно требуют особой обработки: ошибки и альтернативные варианты написания (также известные как альтернативные метки). При вводе запросов пользователи часто делают ошибки в написании своих ключевых слов, и общее ожидание состоит в том,

что поисковая система на базе ИИ сможет понять и правильно обработать эти ошибки.

Хотя общие связанные фразы для «laptop» могут быть «computer», «netbook» или «tablet», ошибки будут больше похожи на «lатор», «laptok» или «lapptop». *Альтернативные метки* функционально ничем не отличаются от ошибок, но возникают, когда существует несколько допустимых вариантов для фразы (например, «specialized» против «specialised» или «cybersecurity» против «cyber security»). В случае как орфографических ошибок, так и альтернативных меток конечной целью обычно является нормирование менее распространенного варианта в более распространенную каноническую форму, а затем поиск канонической версии.

Проверка орфографии может быть реализована несколькими способами. В этом разделе мы рассмотрим встроенную проверку орфографии на основе документов, которая есть в большинстве поисковых систем, а также покажем, как можно извлекать пользовательские сигналы для точной настройки исправлений орфографии на основе реального взаимодействия пользователя с вашей поисковой системой.

6.5.1. Изучение исправлений орфографии из документов

Большинство поисковых систем содержат некоторые встроенные возможности проверки орфографии на основе терминов, найденных в документах коллекции. Например, Apache Solr предоставляет компоненты проверки орфографии на основе файлов, словарей и индексов. Для проверки орфографии на основе файлов требуется составить список терминов, которые можно исправить. Проверка орфографии на основе словаря может создать список терминов для исправления орфографии из полей в индексе. Проверка орфографии на основе индекса может использовать поле в основном индексе для проверки орфографии напрямую, без необходимости создания отдельного индекса проверки орфографии. Кроме того, если кто-то создал список исправлений орфографии в автономном режиме, вы можете использовать список синонимов, чтобы напрямую заменить или расширить любые орфографические ошибки до их канонической формы.

Elasticsearch и OpenSearch имеют схожие возможности проверки орфографии, даже позволяя определенным контекстам уточнять область предложений по орфографии для определенной категории или географической локации.

Хотя мы призываем вас протестировать эти готовые алгоритмы проверки орфографии, все они, к сожалению, страдают от серьезной проблемы: отсутствия пользовательского контекста. В частности, всякий раз, когда ищется ключевое слово, которое не появляется в индексе минимальное количество раз, компонент проверки орфографии начинает просматривать все термины в индексе, которые *отличаются на минимальное количество символов*, а затем возвращает наиболее распространенные ключевые слова в индексе, которые соответствуют критериям. В следующем списке показан пример того, как стандартная конфигурация проверки орфографии на основе индекса терпит неудачу.

Листинг 6.13. Использование готовых исправлений орфографии в документах

```
products_collection = engine.get_collection("products")
query = "moden"
results = engine.spell_check(products_collection, query)
print(results)
```

Вывод:

```
{'modes': 421, 'model': 159, 'modern': 139, 'modem': 56, 'mode6': 9}
```

В листинге 6.13 вы можете увидеть запрос пользователя для `moden`. Проверка орфографии возвращает предлагаемые исправления орфографии – «modes», «model», «modern» и «modem», а также одно предложение, которое появляется только в нескольких документах и которое мы проигнорируем. Поскольку наша коллекция – это технические продукты, может быть очевидно, какое из них, скорее всего, является лучшим исправлением орфографии: это «modem». Фактически маловероятно, что пользователь будет намеренно искать «modes» или «model» как отдельные запросы, поскольку это общие термины, которые обычно имеют смысл только в контексте, содержащем другие слова.

Индекс на основе контента не может легко отличить, что конечные пользователи вряд ли будут искать «modern» или «model». Таким образом, хотя проверки орфографии на основе контента могут хорошо работать во многих случаях, часто можно точнее узнать исправления орфографии из поведения пользователей при поиске.

6.5.2. Изучение исправлений орфографии по сигналам пользователя

Возвращаясь к нашему основному тезису из раздела 6.3 о том, что пользователи склонны искать связанные запросы, пока не найдут ожидаемые результаты, следует предположить, что пользователь, который неправильно написал определенный запрос и получил плохие результаты, затем попытается исправить свой запрос.

Мы уже знаем, как находить связанные фразы (обсуждалось в разделе 6.3), но в этом разделе мы рассмотрим, как конкретно отличить неправильное написание на основе сигналов пользователя. Эта задача в основном сводится к двум целям:

- найти термины с похожим написанием;
- выяснить, какой термин является правильным написанием, а какой – неправильно написанным.

Для этой задачи мы будем полагаться исключительно на сигналы запроса. Мы выполним некоторое предварительное нормирование, чтобы сделать анализ запроса нечувствительным к регистру и отфильтровать дублирующиеся запросы, чтобы избежать спама сигнала. (Мы обсудим нормирование сигнала в разделах 8.2-8.3.) В следующем листинге показан запрос, который захватывает наши нормированные сигналы запроса.

Листинг 6.14. Получение всех запросов, найденных пользователями

```
def get_search_queries():
    query = """SELECT searches.user AS user,
                LOWER(TRIM(searches.target)) As query
                FROM signals AS searches WHERE searches.type = 'query'
                GROUP BY searches.target, user"""
    return spark.sql(query).collect()
```

Строчные буквы в запросах заставляют анализ запроса игнорировать варианты с заглавными буквами.

Группирование по пользователю предотвращает спам от того, что один и тот же пользователь вводит один и тот же запрос много раз.

Для целей этого раздела мы предположим, что запросы могут содержать несколько разных ключевых слов и что мы хотим рассматривать каждое из этих ключевых слов как потенциальный вариант написания. Это позволит находить отдельные термины и заменять их в будущем запросе, в отличие от рассмотрения всего запроса как одной фразы. Это также позволит нам отбросить определенные термины, которые, вероятно, будут шумом, например стоп-слова или стоящие отдельно числа.

Следующий листинг демонстрирует процесс токенизации каждого запроса для создания списка слов, на основе которого мы можем провести дальнейший анализ.

Листинг 6.15. Поиск слов путем токенизации и фильтрации терминов запроса

```
from nltk import tokenize, corpus, download
download('stopwords')
stop_words = set(
    corpus.stopwords.words("english"))

def is_term_valid(term, minimum_length=4):
    return (term not in stop_words and
            len(term) >= minimum_length and
            not term.isdigit())

def tokenize_query(query):
    return tokenize.RegexpTokenizer(r'\w+').tokenize(query)

def valid_keyword_occurrences(searches, tokenize=True):
    word_list = defaultdict(int)
    for search in searches:
        query = search["query"]
        terms = tokenize_query(query) if tokenize else [query]
        for term in terms:
            if is_term_valid(term):
                word_list[term] += 1
    return word_list
```

Определяет стоп-слова, которые не следует считать опечатками или исправлениями, используя Natural Language Toolkit (nltk).

Удаляет шумные термины, включая стоп-слова, очень короткие термины и числа.

Разбивает запрос пробелами на отдельные термины при токенизации.

Объединяет вхождения допустимых ключевых слов.

После очистки списка токенов следующим шагом будет отделение токенов с высокой частотой от токенов с низкой частотой. Поскольку орфографические ошибки будут встречаться относительно редко, а правильные – чаще, мы будем использовать относительное количество вхождений, чтобы определить, какая версия, скорее всего, является каноническим написанием, а какие варианты – орфографическими ошибками.

Чтобы обеспечить максимальную чистоту нашего списка исправлений орфографии, мы установим некоторые пороговые значения для популярных терминов и некоторые для терминов с низкой частотой, которые, скорее всего, являются орфографическими ошибками. Поскольку некоторые коллекции могут содержать сотни документов, а другие – миллионы, мы не можем просто смотреть на абсолютное число для этих пороговых значений, поэтому вместо этого будем использовать квантили¹. В следующем листинге показаны расчеты для каждого из квантилей в диапазоне от 0.1 до 0.9.

Листинг 6.16. Расчет квантилей для определения кандидатов на орфографию

```
def calculate_quantiles(word_list):
    quantiles_to_check = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
    quantile_values = numpy.quantile(numpy.array(list(word_list.
values()))),
                                quantiles_to_check)
    return dict(zip(quantiles_to_check, quantile_values))

query_signals = get_search_queries()
word_list = valid_keyword_occurrences(query_signals, tokenize=True)
quantiles = calculate_quantiles(word_list)
display(quantiles)
```

Вывод:

```
{0.1: 5.0,
 0.2: 6.0,
 0.3: 8.0,
 0.4: 12.0,
 0.5: 16.0,
 0.6: 25.0,
 0.7: 47.0,
 0.8: 142.20000000000007,
 0.9: 333.20000000000007}
```

¹ Квантиль – это значение, которое определяет, сколько значений в распределении находятся выше или ниже определенного предела. Например, квартили делят набор данных на четыре равные части, каждая из которых содержит 25 % данных. В общем случае используют три квартиля: Q1 (первый квартиль), Q2 (второй квартиль) и Q3 (третий квартиль) – значения, ниже которых находятся 25, 50 и 75 % данных соответственно. – *Прим. ред.*

Здесь мы видим, что 80 % терминов находятся 142.2 раза или меньше. Аналогично только 20 % терминов находятся 6.0 раз или меньше. Используя принцип Парето, предположим, что большинство наших опечаток попадают в 20 % самых низких по поиску терминов, а большинство наших самых важных терминов попадают в 20 % самых высоких по поиску запросов. Если вам нужна более высокая точность (генерировать исправления орфографии только для высокочастотных терминов и только если вероятность ложных срабатываний низкая), вы можете переместить их в квантиль 0.1 для опечаток и квантиль 0.9 для правильно написанных терминов. Вы также можете пойти в другом направлении, чтобы попытаться сгенерировать больший список опечаток с более высокой вероятностью ложных срабатываний.

В листинге 6.17 мы разделим термины на группы, назначив низкочастотные термины группе опечаток, а высокочастотные термины группе исправлений. Эти группы станут отправной точкой для поиска высококачественных исправлений орфографии, когда достаточное количество пользователей будут искать как кандидата с ошибкой, так и кандидата на исправление.

Листинг 6.17. Определение кандидатов на исправление орфографии

```
def create_spelling_candidates(word_list):
    quantiles = calculate_quantiles(word_list)
    misspellings = {"misspelling": [],
                    "misspell_counts": [],
                    "misspell_length": [],
                    "initial": []}
    corrections = {"correction": [],
                  "correction_counts": [],
                  "correction_length": [],
                  "initial": []}
    for key, value in word_list.items():
        if value <= quantiles[0.2]:
            misspellings["misspelling"].append(key)
            misspellings["misspell_counts"].append(value)
            misspellings["misspell_length"].append(len(key))
            misspellings["initial"].append(key[0])
        if value >= quantiles[0.8]:
            corrections["correction"].append(key)
            corrections["correction_counts"].append(value)
            corrections["correction_length"].append(len(key))
            corrections["initial"].append(key[0])
    return (pandas.DataFrame(misspellings), pandas.DataFrame(corrections))
```

Первая буква термина сохраняется для ограничения области проверки ошибок.

Термины, находящиеся на уровне или ниже квантиля 0.2, добавляются в список ошибок.

Количество поисков сохраняется для отслеживания популярности.

Длина термина будет использоваться позже для установки пороговых значений для расчетов расстояния редактирования.

Для верхних 20 % терминов сохраняются те же данные, но в списке исправлений.

Чтобы эффективно сравнить все misspellings и значения corrections, мы сначала загружаем их в датафреймы¹ в листинге 6.17. Вы можете

¹ Датафрейм – это тип данных, используемый для работы с табличными данными. Его можно представить в виде обычной таблицы с любым количеством столбцов и строк. Внутри ячеек такой «таблицы» могут быть данные самого разного типа: числовые,

себе представить, что `corrections` – это безупречный список самых популярных искомым терминов, в то время как список `misspellings` должен предоставить хороший список кандидатов для терминов, которые ищут реже, они с большей вероятностью являются опечатками.

Когда мы сравниваем опечатки с кандидатами на правильное написание и решаем, сколько различий в символах (или *расстояний редактирования*) допустимо, нам нужно учитывать длину термина. В следующем листинге показана простая функция `good_match`, которая определяет общую эвристику для того, на сколько расстояний редактирования может отличаться соответствие термина, при этом опечатка по-прежнему считается вероятным кандидатом на исправление.

Листинг 6.18. Поиск правильных написаний по длинам и расстоянию редактирования

```
def good_match(word_length_1, word_length_2, edit_dist):
    min_length = min(word_length_1, word_length_2)
    return ((min_length < 8 and edit_dist == 1) or
            (min_length >= 8 and min_length < 11 and edit_dist <= 2) or
            (min_length >= 11 and edit_dist == 3))
```

После загрузки наших `misspellings` и кандидатов на `corrections` в датафреймы и определения функции `good_match` пришло время сгенерировать наш список исправлений орфографии. Как и в разделе 6.5.1, где исправления орфографии были сгенерированы на основе расстояний редактирования и количества вхождений терминов в нашей коллекции документов, код в листинге 6.19 генерирует исправления орфографии на основе расстояний редактирования и вхождений терминов в наших журналах запросов.

Листинг 6.19. Сопоставление ошибок с их правильными написаниями

```
from nltk import edit_distance

def calculate_spelling_corrections(word_list):
    (misspellings, corrections) = create_spelling_candidates(word_list)
    matches_candidates = pandas.merge(misspellings,
                                     corrections, on="initial")
    matches_candidates["edit_dist"] = matches_candidates.apply(
        lambda row: edit_distance(row.misspelling,
                                  row.correction), axis=1)
```

Группирует орфографические ошибки и кандидаты на исправление по первой букве слова.

Вычисляет расстояние редактирования между каждой орфографической ошибкой и кандидатом на исправление.

булевы, строковые и т. д. Датафрейм состоит из колонок и строк, где в колонках указаны переменные, а в строках – значения, соответствующие каждой переменной. У датафрейма есть и индексы строк, и индексы столбцов, что позволяет удобно сортировать и фильтровать данные, а также быстро находить нужные ячейки. – Прим. ред.

```
matches_candidates["good_match"] = matches_candidates.apply(
    lambda row: good_match(row.misspell_length,
                           row.correction_length,
                           row.edit_dist),axis=1)
```

Применяет функцию `good_match`, используя длину терминов и расстояние редактирования.

```
cols = ["misspelling", "correction", "misspell_counts",
        "correction_counts", "edit_dist"]
matches = matches_candidates[matches_candidates["good_match"]] \
    .drop(["initial", "good_match"],axis=1) \
    .groupby("misspelling").first().reset_index() \
    .sort_values(by=["correction_counts", "misspelling",
                    "misspell_counts"],
                ascending=[False, True, False])[cols]
return matches
```

Объединяет все орфографические ошибки по имени.

```
query_signals = get_search_queries()
word_list = valid_keyword_occurrences(query_signals, tokenize=True)
corrections = calculate_spelling_corrections(word_list)
display(corrections.head(20))
```

Получает 20 слов с наибольшим количеством ошибок.

Вывод:

	misspelling	correction	misspell_counts	correction_counts	edit_dist
50	iphone3	iphone	6	16854	1
61	laptopa	laptop	6	14119	1
62	latop	laptop	5	14119	1
...					
76	moden	modem	5	3590	1
77	modum	modem	6	3590	1
135	tosheba	toshiba	6	3432	1
34	gates	games	6	3239	1
84	phono	phone	5	3065	1

Как вы можете видеть, теперь у нас есть относительно чистый список исправлений орфографии на основе сигналов пользователя. Наш запрос `moden` правильно сопоставляется с «`modem`», в отличие от маловероятных терминов поиска, таких как «`model`» и «`modern`», которые мы видели в исправлении орфографии на основе документа в листинге 6.13.

Существует множество других способов, которыми вы могли бы создать модель исправления орфографии. Если вы хотите сгенерировать исправления орфографии для нескольких терминов из документов, можете сгенерировать биграммы и триграммы для выполнения цепного байесовского анализа вероятностей появления последовательных терминов. Аналогично, чтобы генерировать многотерминные исправления орфографии из сигналов запросов, вы можете удалить токенизацию запросов, установив `tokenize` в `False` при вызове `valid_keyword_occurrences`.

Листинг 6.20. Поиск многотерминных исправлений орфографии из полных запросов

```
query_signals = get_search_queries()
word_list = valid_keyword_occurrences(query_signals, tokenize=False)
corrections = calculate_spelling_corrections(word_list)
display(corrections.head(20))
```

Вывод:

misspelling	correction	misspell_counts	correction_
counts	edit_dist		
181 ipad.	ipad	6	7749
154 hp touchpad 32	hp touchpad	5	7144
155 hp toucpad	hp touchpad	6	7144
153 hp tochpad	hp touchpad	6	7144
190 iphone s4	iphone 4s	5	4642
193 iphone4 s	iphone 4s	5	4642
194 iphones 4s	iphone 4s	5	4642
412 touchpaf	touchpad	5	4019
406 tochpad	touchpad	6	4019
407 toichpad	touchpad	6	4019
229 latop	laptop	5	3625
228 laptopa	laptops	6	3435
237 loptops	laptops	5	3435
205 ipods touch	ipod touch	6	2992
204 ipod tuch	ipod touch	6	2992
165 i pod tuch	ipod touch	5	2992
173 ipad 2	ipad 2	6	2807
215 kindle	kindle	5	2716
206 ipone	iphone	6	2599
192 iphone3	iphone	6	2599

Вы можете увидеть некоторые из распространенных многотерминных орфографических ошибок и их исправления в листинге 6.20, где запросы больше не токенизируются. Обратите внимание, что одотерминные слова в основном одинаковы, но многотерминные запросы также были проверены на орфографию. Это отличный способ нормализовать названия продуктов, так что все запросы «iphone4 s», «iphones 4s» и «iphone s4» правильно отображаются в каноническом «iphone 4s». Обратите внимание, что в некоторых случаях это может быть процесс с потерями, так как «hp touchpad 32» отображается в «hp touchpad», а «iphone3» отображается в «iphone». В зависимости от вашего варианта использования, вам может быть полезно только исправить орфографию отдельных терминов или включить специальную обработку в вашу функцию `good_match` для вариаций бренда, чтобы гарантировать, что код проверки орфографии не удалит по ошибке релевантный контекст запроса.

6.6. Собираем все вместе

В этой главе мы глубже погрузились в понимание контекста и значения домен-специфического языка. Мы показали, как использовать SKG для классификации запросов и устранения неоднозначности терминов, которые имеют разные или нюансные значения на основе их контекста. Мы также изучили, как извлекать связи из пользовательских сигналов, что обычно обеспечивает лучший контекст для понимания ваших пользователей, чем просмотр только ваших документов. Мы также показали, как извлекать фразы, орфографические ошибки и альтернативные метки из сигналов запроса, что позволяет изучать домен-специфическую терминологию непосредственно от пользователей, а не только из документов.

На этом этапе вы должны быть уверены в изучении домен-специфичных фраз и связанных фраз из документов или пользовательских сигналов, классификации запросов к доступному контенту и устранении неоднозначности значения терминологии на основе классификации запроса. Эти методы являются критически важными инструментами в вашем наборе инструментов для интерпретации намерения запроса.

Наша цель – не просто собрать большой набор инструментов. Наша цель – использовать каждый из этих инструментов там, где это уместно, для построения сквозного семантического слоя поиска. Это означает, что нам нужно встраивать известные фразы в наш граф знаний, извлекать их из входящих запросов, обрабатывать опечатки, классифицировать запросы, устранять неоднозначность входящих терминов и в конечном итоге генерировать переписанный запрос для поисковой системы, которая использует каждый из наших методов поиска на основе ИИ. В следующей главе мы покажем вам, как собрать каждый из этих методов в работающую систему семантического поиска, предназначенную для наилучшей интерпретации и моделирования намерения запроса.

Резюме

- Классификация запросов с использованием семантического графа знаний (SKG) может помочь интерпретировать намерение запроса и улучшить маршрутизацию и фильтрацию запросов.
- Разрешение неоднозначности в смысле запроса может обеспечить более контекстное понимание запроса пользователя, особенно для терминов со значительно различающимися в разных контекстах значениями.
- Помимо обучения на основе документов, домен-специфичные и связанные фразы также можно изучать на основе сигналов пользователя.
- Ошибки и вариации написания можно изучать как на основе документов, так и на основе сигналов пользователя, при этом подходы на основе документов являются более надежными, а подходы на основе сигналов пользователя лучше отражают его намерения.



Интерпретация намерения запроса через семантический поиск

В этой главе рассматривается:

- механика интерпретации запроса;
- реализация сквозного конвейера намерения запроса для парсинга, обогащения, преобразования и поиска;
- тегирование и классификация терминов и фраз запроса;
- аугментация¹ запросов с помощью обходов графа знаний;
- интерпретация семантики шаблонов домен-специфичных запросов.

В главах 5 и 6 мы использовали контент и сигналы для интерпретации домен-специфичного смысла входящих пользовательских запросов. Мы обсуждали идентификацию фраз, обнаружение опечаток, обнаружение синонимов, классификацию намерения запроса, расширение

¹ Аугментация в программировании – это метод увеличения выборки данных для обучения через модификацию этих данных и создание на этой основе дополнительных. Он используется в машинном обучении, например для обучения нейронных сетей, когда исходный обучающий набор содержит ограниченное количество данных. – *Прим. ред.*

связанных терминов и даже устранение неоднозначности смысла запроса. Мы в основном обсуждали эти методы изолированно, чтобы продемонстрировать, как каждый из них работает независимо от другого.

В этой главе мы применим все эти методы на практике, интегрировав их в единую структуру интерпретации запроса. Мы покажем пример интерфейса поиска, который принимает реальные запросы, интерпретирует их, переписывает их, чтобы лучше выразить намерение конечного пользователя, а затем возвращает ранжированные результаты.

Мы должны отметить, что для реализации семантического поиска было разработано несколько парадигм, включая интерпретацию запросов на основе эмбединга и ответы на вопросы (возвращение извлеченных или сгенерированных ответов вместо документов) с использованием больших языковых моделей (LLM) и предварительно обученных преобразователей. Они обычно включают кодирование запросов в векторы, поиск приблизительного ближайшего соседства векторов, а затем выполнение расчета сходства векторов для ранжирования документов. Ранжированные документы затем часто анализируются для обобщения, извлечения или генерации ответов. Мы рассмотрим эти подходы на основе LLM для семантического поиска и ответов на вопросы в главах 13–15.

В этой главе мы сосредоточимся на механизмах интеграции каждой из стратегий поиска на основе ИИ, которые вы уже изучили, для предоставления сквозного конвейера семантических запросов. Мы реализуем конвейер в четыре этапа:

- парсинг запроса пользователя;
- обогащение проанализированного запроса улучшенным контекстом;
- преобразование запроса для оптимизации релевантности в нашей целевой поисковой системе;
- поиск с использованием оптимизированного запроса.

Эти шаги не обязательно должны быть реализованы линейно (иногда они повторяются, а иногда их можно пропустить), также их можно разбить на более мелкие части (например, поиск можно разбить на сопоставление, ранжирование и реранкинг). Тем не менее наличие этой последовательной структуры, с помощью которой мы можем интегрировать любую комбинацию методов поиска на основе ИИ, будет бесценным, поскольку вы смешиваете и сопоставляете подходы в своих собственных поисковых приложениях.

7.1. Механика интерпретации запроса

Не существует единственного «правильного способа» создания структуры интерпретации запроса. Каждая организация, создающая интеллектуальную поисковую платформу, вероятно, создаст что-то немного другое в зависимости от потребностей своего бизнеса и опыта своей поисковой команды. Однако есть некоторые общие темы в реализациях, которые стоит изучить.

- Конвейеры – как при индексации документов, так и при обработке запросов полезно моделировать всю необходимую логику синтаксического анализа, интерпретации и ранжирования как модульные этапы в рабочем процессе. Это позволяет легко экспериментировать, заменяя, перестраивая или добавляя этапы обработки в конвейере в любое время.
- Модели – независимо от того, настраиваете ли вы сложный LLM на основе глубокого обучения (главы 13–14), модель обучения ранжированию (главы 10–12), модель бустинга сигналов или персонализации (главы 8–9) или граф знаний, содержащий синонимы, орфографические ошибки и связанные термины (главы 5–6), правильная интерпретация запросов требует подключения правильных моделей в правильном порядке в конвейерах индексации и запросов.
- Условные откаты – вы никогда не сможете идеально интерпретировать каждый запрос. У вас может быть много моделей, помогающих интерпретировать один запрос, без малейшего представления о том, что означает другой запрос. Обычно лучше всего начать с базовой или «резервной» модели (обычно на основе ключевых слов), которая может несовершенно обработать любой запрос, и наложить более сложные модели поверх, чтобы повысить точность интерпретации. Кроме того, если результаты не найдены, может быть полезно вернуть рекомендации, чтобы гарантировать, что пользователь увидит что-то полезное, даже если это не совсем то, что он искал.

На рис. 7.1 показан пример конвейера запросов, демонстрирующий каждую из этих тем объединения этапов конвейера, моделей и условных резервов.

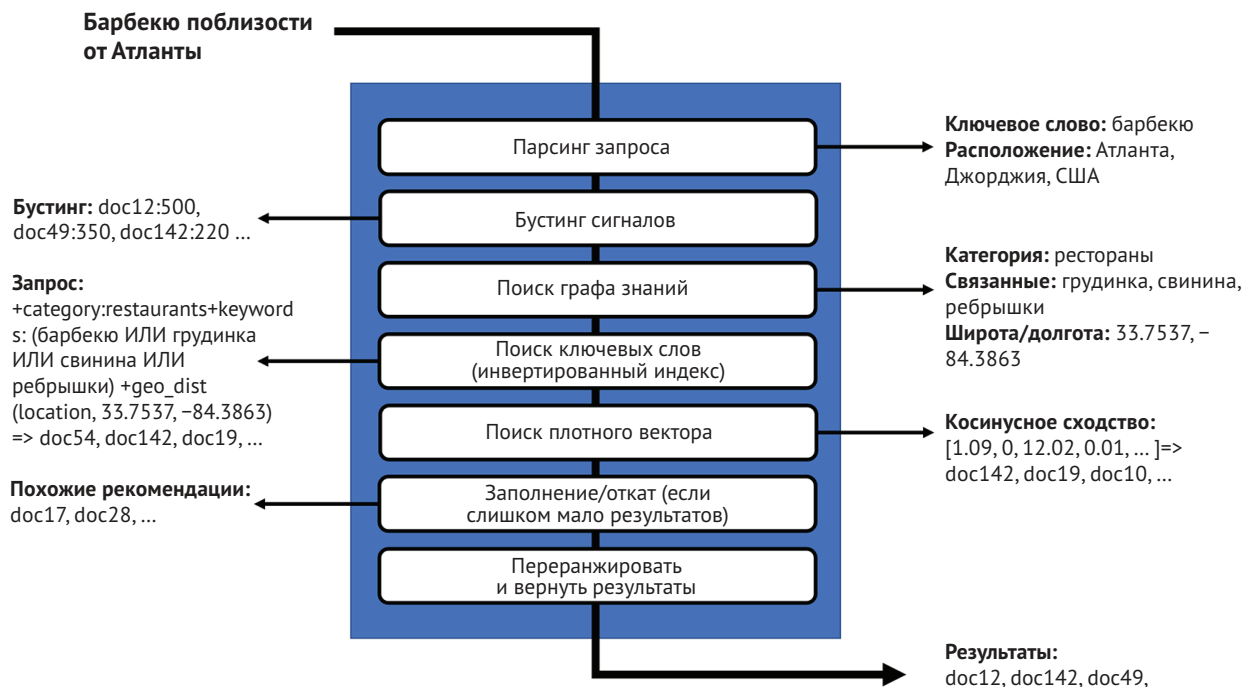


Рис. 7.1. Пример конвейера интерпретации запроса

В рис. 7.1 принимается запрос `bbq near atlanta` и начинается его обработка с этапа синтаксического анализа (парсинга), который выполняет извлечение сущностей из запроса по известным ключевым словам, локациям или другим известным терминам. Затем он достигает этапа бустинга сигналов и проверяется с помощью модели бустинга сигналов (которая была представлена в главе 4 и будет рассмотрена более подробно в главе 8), чтобы поднять самые популярные документы для данного запроса. Для интерпретации отдельных ключевых слов и связывания их друг с другом обычно используются три различных, но взаимодополняющих подхода, все из которых включены в пример конвейера:

- лексический поиск, такой как ранжирование BM25 по совпадениям булевых запросов в инвертированном индексе;
- поиск по графу знаний, такой как ранжирование сущностей, найденных в запросе, и их связей с наиболее похожими сущностями в вашем индексе с использованием семантического графа знаний (SKG) или явно построенного графа знаний;
- поиск плотных векторов, такой как косинусное сходство векторов, найденных с приближительной ближайшей окрестностью эмбедингов.

Из этих трех словев наиболее распространенным слоем сопоставления «по умолчанию» является лексический поиск по инвертированному индексу, поскольку этот подход позволяет делать сопоставление *любого* термина, который существует в корпусе документов, независимо от того, понятен этот термин или нет. Подходы графа знаний и плотных векторов основаны на возможности связывать термины в каждом запросе с понятиями или сущностями, и это просто невозможно сделать во всех случаях.

Фактически ранжирование BM25 часто превосходит подходы плотных векторов к эмбедингу даже из самых современных LLM, если только эти языковые модели изначально не обучены или не настроены на домен-специфический контент (это может измениться со временем, поскольку предварительно обученные LLM продолжают становиться все более надежными). Мы погрузимся в использование LLM, начиная с глав 9 и 13, для персонализированного поиска и семантического поиска, и мы потратим время на тонкую настройку и использование LLM для более продвинутых функций поиска, таких как ответы на вопросы и генеративный поиск, в главах 14 и 15. В этой главе мы сосредоточимся в первую очередь на демонстрации механизмов интеграции лексического поиска и графов знаний.

Конвейер на рис. 7.1 заканчивается этапом обратного заполнения/отката, который может быть полезен для эмбединга дополнительных результатов в случае, если ни один из предыдущих этапов не смог вернуть полный набор результатов. Это может быть так же просто, как возврат рекомендаций вместо результатов поиска (рассматривается в главе 9), или это может включать возврат частично совпавшего запроса с более низкой точностью.

Далее конечные результаты всех этапов конвейера объединяются и переранжируются по мере необходимости для получения окончательного набора результатов поиска, ранжированных по релевантности. Этап реранкинга может быть простым, но он часто будет реализован с использованием машинного ранжирования через классификатор ранжирования. Мы углубимся в создание и автоматизацию обучения машинно-обученному ранжированию моделей в главах 10–12.

Хотя пример конвейера в этом разделе может давать хорошие результаты во многих случаях, конкретная логика конвейера всегда должна зависеть от потребностей вашего приложения. В следующем разделе мы настроим приложение для поиска отзывов о местных предприятиях, а затем реализуем унифицированный конвейер, способный выполнять семантический поиск по этому домену.

7.2. Индексирование и поиск в наборе данных локальных отзывов

Мы собираемся создать поисковую систему, которая агрегирует отзывы о продуктах и предприятиях со всего интернета. Если у компании есть физическая локация (ресторан, магазин и т. д.), мы хотим найти все отзывы, связанные с этой компанией, и сделать их доступными для поиска.

В следующем листинге показано, как наши данные сканирования локальных отзывов попадают в поисковую систему.

Листинг 7.1. Загрузка и индексация набора данных отзывов

```
reviews_collection = engine.create_collection("reviews")
reviews_data = reviews.load_dataframe("data/reviews/reviews.csv")
reviews_collection.write(reviews_data)
```

Вывод:

```
Wiping "reviews" collection
Creating "reviews" collection
Status: Success
```

Loading Reviews...

```
root
|-- id: string (nullable = true)
|-- business_name: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- content: string (nullable = true)
|-- categories: string (nullable = true)
|-- stars_rating: integer (nullable = true)
|-- location_coordinates: string (nullable = true)
```

Successfully written 192138 documents

Модель данных для этих отзывов можно увидеть в выводе листинга. Каждый отзыв содержит название компании, информацию о локации, содержание отзыва, рейтинг отзыва (количество звезд от 1 до 5) и категории типа рассматриваемой сущности.

После того как данные будут получены, мы можем запустить поиск. В этой главе мы предоставим более интерактивное приложение, чем в предыдущих главах, запустив веб-сервер для поддержки динамического интерфейса поиска. Запуск листинга 7.2 запустит веб-сервер.

Листинг 7.2. Запуск веб-сервера и загрузка страницы поиска

```
start_reviews_search_webserver()  
  
%%html  
<iframe src="http://localhost:2345/search" width=100% height="800"/>
```

На рис. 7.2 показан загруженный интерфейс поиска из листинга 7.2. Вы можете запустить встроенную страницу поиска из блокнота Jupyter, но если вы работаете на локальном компьютере через порт 2345, вы также можете просто перейти по адресу <http://localhost:2345/search> в своем веб-браузере, чтобы получить более удобный интерфейс.

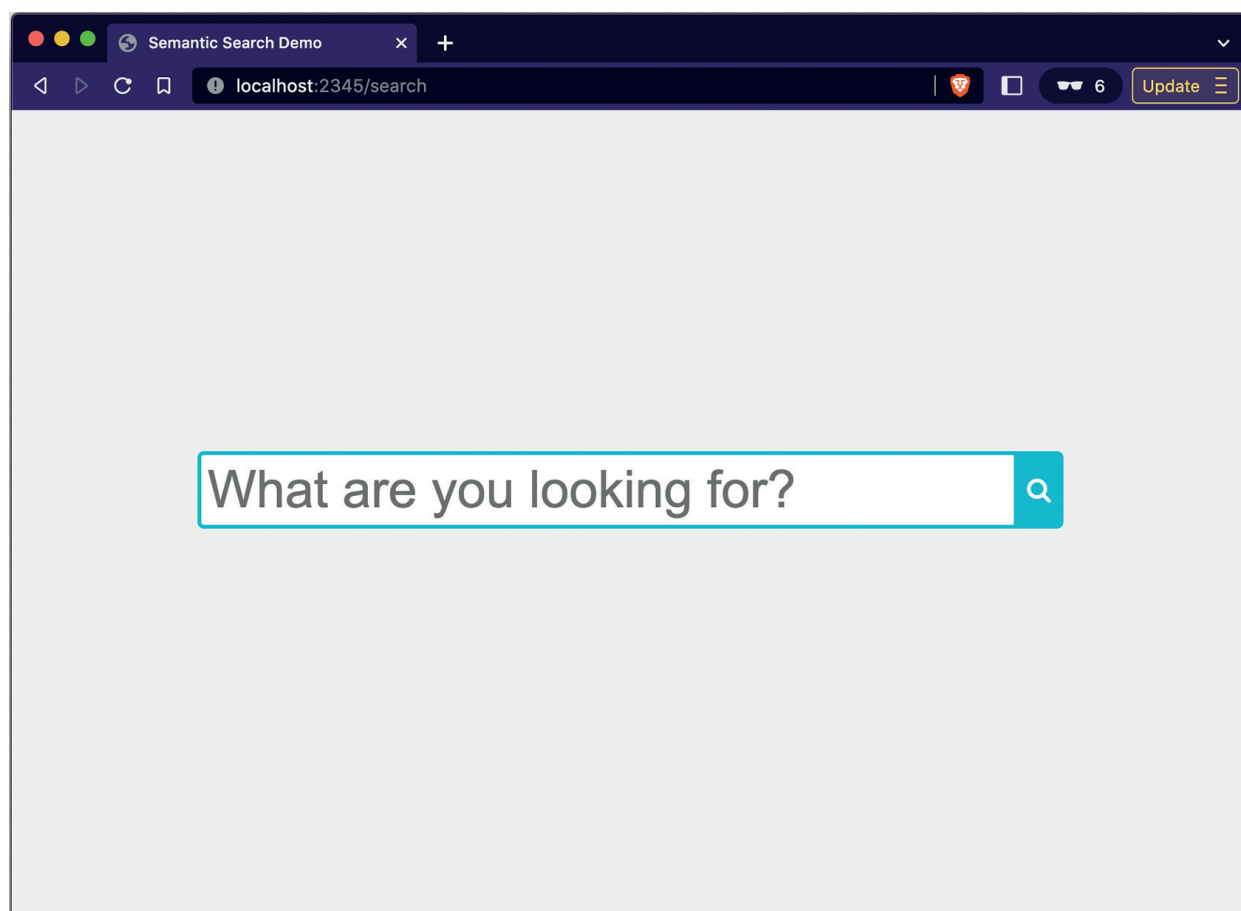


Рис. 7.2. Посещение страницы поиска отзывов из локального веб-браузера

Сначала попробуем простой запрос – `bbq near charlotte`. Пока что давайте представим, что вы не прошли процесс обучения графа знаний

(глава 6) и еще не знаете, как применить SKG (глава 5) к интерпретации вашего запроса. В этом случае мы просто выполняем нестандартное лексическое сопоставление ключевых слов. На рис. 7.3 показан верхний лексический результат поиска для запроса `bbq near charlotte`.

bbq near charlotte



Name: Mint Hill Madness | **City:** Mint Hill, NC | **Rating:** ★★★★★

Mint Hill Madness is one of the older festivals around town, beat out only by a few like the popular Festival in the Park in **Charlotte**. The rides are **near** City Hall and you can get an all day pass for all the rides. Prices are very reasonable - rather than high prices for a barbecue sandwich at some festivals (like **BBQ** and Blues), you pay about half as much. There's also pizza by the slice, kettle corn, Italian ice, and other general festival food. There are also bands playing by City Hall at night!

Рис. 7.3. Базовый лексический поиск ключевых слов для `bbq near charlotte`, только соответствующие запросу ключевые слова

В нашем наборе данных отзывов это единственный отзыв, который соответствует нашему запросу, хотя в городе Шарлотт, Северная Каролина, США, или рядом с ним есть несколько ресторанов барбекю (BBQ). Причина, по которой возвращается только этот результат, заключается в том, что это единственный документ, содержащий все три термина (`bbq`, `near` и `charlotte`). Если вы посмотрите на отзыв, то он даже не для ресторана, где подают барбекю, а на самом деле для фестиваля, в обзоре которого случайно упоминается другой фестиваль с «BBQ» в названии!

Главная проблема здесь в том, что большинство соответствующих ресторанов не содержат слова `near`. Рисунок 7.4 показывает, что мы получим больше результатов, если уберем слово `near` и вместо этого запросим `bbq charlotte`.

bbq charlotte



Name: Boardwalk Billy's Raw Bar And Ribs | **City:** Charlotte, NC | **Rating:** ★

Service was awesome. The food on the other hand was a complete joke! Portions were small for how much they charge. I hope this was not the only BBQ representing Charlotte! Again the service was awesome but when I leave still hungry and spend over \$10 bucks there is a problem.



Name: Grandma's Country Kitchen | **City:** Charlotte, NC | **Rating:** ★★★★★

Phenomenal hole in the wall. I'm a sucker for soul food and this place is legit. One of the first restaurants I visited after moving to Charlotte from Oregon and Grandma's delivers every time. BBQ Chicken is a favorite as are the wings (lot of flavor, not a lot of grease). Mac n cheese, collards, and yams are my standard sides but everything has been delicious with beautiful consistency

Рис. 7.4. Базовый лексический поиск ключевых слов для `bbq charlotte`. Больше соответствующих запросу результатов после удаления из него слова «near»

Оба топовых результата содержат термин «bbq», но первый имеет низкий рейтинг (1 звезда), а во втором упоминается «bbq chicken» (курица с соусом барбекю), но не «bbq» (барбекю), что обычно относится к копченому мясу, такому как свинина, курица, ребра или грудинка. Кроме того, хотя все результаты находятся в городе Шарлотт, Северная Каролина, это происходит только потому, что они соответствуют ключевому слову *charlotte* в тексте обзора, что означает, что отсутствуют многие хорошие результаты, которые не ссылались на город по названию в обзоре. Из результатов ясно, что поисковая система неточно интерпретировала намерение запроса пользователя.

Мы можем сделать гораздо лучше! Вы уже научились извлекать домен-специфичные знания и классифицировать запросы (например, *bbq* подразумевает ресторан), поэтому нам просто нужно интегрировать эти методы и изученные модели сквозным образом.

7.3. Пример сквозного семантического поиска

В последнем разделе были показаны недостатки использования чистого поиска по ключевым словам. Как мы можем улучшить способность поисковой системы интерпретировать запросы? На рис. 7.5 показаны результаты достаточно конкретного запроса, который традиционный поиск по ключевым словам с трудом мог бы правильно интерпретировать: *top kimchi near charlotte*.

The screenshot shows a search interface with the query "top kimchi near charlotte" in a search bar. Below the bar, the results are displayed in three sections:

- TAGGED:** {top} kimchi {near} {charlotte}
- PARSED:** A JSON object containing semantic functions and their parameters. It identifies 'top' as a popularity function, 'kimchi' as a keyword, and 'near charlotte' as a location distance function.
- ENRICHED & TRANSFORMED:** A complex query string for a database, incorporating popularity ratings, location coordinates, and other semantic information.

Рис. 7.5. Семантический поиск для запроса *top kimchi near charlotte*

Этот запрос интересен, потому что он содержит только одно ключевое слово («kimchi») для целей релевантного ранжирования. Ключевое слово «top» на самом деле означает «самый популярный» или «самый высоко оцененный», а фраза «near charlotte» указывает на географический фильтр для результатов поиска. Вы можете видеть на рисунке, что исходный запрос анализируется как {top} kimchi {near} {charlotte}. Мы используем этот синтаксис с фигурными скобками, чтобы указать, что термины «top», «near» и «charlotte» были идентифициро-

ваны из нашего графа знаний, тогда как «kimchi» не был помечен и, следовательно, является неизвестным.

После анализа этих ключевых слов и фраз вы увидите, что они обогащены и преобразованы в следующий синтаксис, специфичный для поисковой системы (Solr):

- `top: +{!func v="mul(if(stars_rating,stars_rating,0),20)"}`. Этот синтаксис повысит рейтинг всех документов на основе их отзывов (от 1 до 5 звезд), умножая на 20, чтобы получить оценку от 0 до 100;
- `kimchi: +{!edismax v="kimchi^0.9193 korean^0.7069 banchan^0.6593+doc_type:\"Korean\""}`. Это расширение неизвестного термина «кимчи» с использованием подхода SKG к расширению из главы 5. В этом случае SKG определяет «корейский» как категорию, в которой следует фильтровать результаты, а наиболее часто встречающиеся термины для «kimchi» – «korean» и «banchan»;
- `near charlotte: +{!geofilt d=50 sfield="location_coordinates" pt="35.22709,-80.84313"}`. Этот географический фильтр ограничивает результаты документами только в радиусе 50 км от широты/долготы Шарлотта, Северная Каролина, США.

Если бы исходный запрос был выполнен как традиционный лексический поиск без слоя интерпретации запроса, то не было бы релевантных результатов, как показано на рис. 7.6.

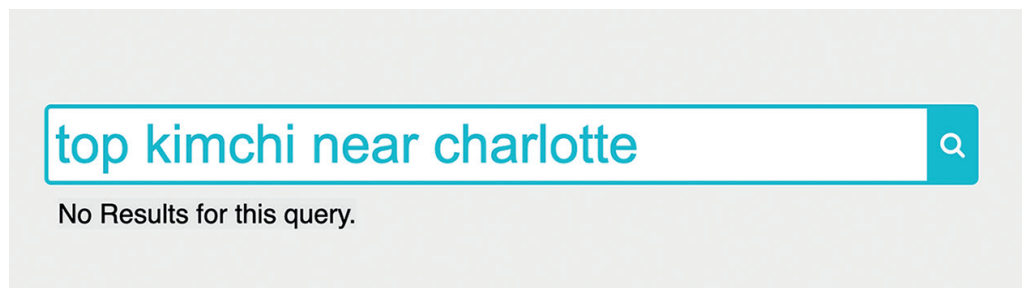


Рис. 7.6. Традиционный лексический поиск не возвращает результатов, поскольку нет документов, содержащих все ключевые слова

Однако рис. 7.7 демонстрирует результаты после выполнения парсинга и обогащения версии.

Результаты выглядят довольно хорошо! Вы видите, что:

- результатов много (вместо нуля);
- все результаты имеют высшие оценки (5 звезд);
- все результаты находятся в городе Шарлотт;
- некоторые результаты совпадают даже без основного ключевого слова («kimchi»), и они явно относятся к корейским ресторанам, в которых подают кимчи, поскольку в обзоре использовались похожие термины.

Мы посвятим оставшуюся часть главы рассмотрению того, как мы можем реализовать этот уровень семантической интерпретации запроса, начав с конвейера интерпретации запроса высокого уровня.



Name: Korean Restaurant | **City:** Charlotte, NC | **Rating:** ★★★★★

Copied my review from Super G over here as well as I didn't realize there was a separate listing for this restaurant. It's that good so I need to repeat it. :) I have been shopping here for a while and love the grocery side. But I love even more the restaurants all along the side of this grocery store. There's a bakery and 2 restaurants with delicious offerings. The bakery, the items there are light, fresh and tasty. I take advantage of their buy 5 and get 1 free deal. The Hong Kong restaurant, their soups are delicious. Their Thai tea is outstanding and seriously addicted to it. Their buns are fluffy and stuffed with great fillings. But my all time favorite has to be the Korean restaurant. I have had their spicy beef cabbage soup which lives up to it's name, SPICY but can't stop eating. (get extra thai tea if you get this dish!) Their bulgogi is flavorful and tender. Their black bean noodle is really good. The bibimbap hot dish, the star of the place. And even all their side dishes they offer in the deli case in front of the restaurant is delish. I love their chives kimchi, cabbage kimchi and fish cakes etc. And no I don't work there nor do I know anyone there. It's just that tasty and good. Oh and their portions...HUGE.



Name: Cho Won Garden | **City:** Charlotte, NC | **Rating:** ★★★★★

Johnny P. Sorry you didnt like the food. The food, service, and prices were great. You said go to pepero because you are immature And you dont know why you said that. This place is a 8/10 if you know Of korean culture. This place is good for any occasion..dates too so dont be afraid to try It out.



Name: Hibiscus | **City:** Charlotte, NC | **Rating:** ★★★★★

We ate here for dinner and had a very tasty meal of bibimbap and bulgogi. Both dishes were done well and tasty. We had a very pleasant waitress who provided great service as well. Overall great! We'll definitely be back.

Рис. 7.7. Пример семантического поиска возвращает релевантные результаты за счет лучшей интерпретации и выполнения запроса

7.4. Конвейеры интерпретации запроса

Хотя нам часто требуется интегрировать несколько моделей и различные подходы к пониманию запроса в конвейер запроса, большинство конвейеров запроса имеют схожий набор высокоуровневых фаз.

- 1 *Парсинг* – извлечение ключевых сущностей и их логических связей из запроса.
- 2 *Обогащение* – формирование понимания контекста запроса, запрашиваемых сущностей и их семантических связей.
- 3 *Преобразование* – переписывание запроса пользователя для поисковой системы для оптимизации отзыва и ранжирования.
- 4 *Поиск* – выполнение преобразованного запроса и возврат ранжированных результатов.

Вы можете рассматривать каждую фазу как отдельный тип этапа конвейера. Как и в примере конвейера в разделе 7.1, некоторые конвейеры могут вызывать несколько этапов для парсинга или обогащения запроса, а некоторые конвейеры могут даже запускать несколько условных поисков и объединять результаты.

В следующих подразделах мы реализуем каждую фазу, чтобы продемонстрировать внутреннюю работу нашего примера сквозного семантического поиска из раздела 7.3.

7.4.1. Парсинг запроса для семантического поиска

Как вы видели в разделе 3.2.5, большинство поисковых систем по ключевым словам по умолчанию выполняют некоторую форму булевого анализа входящих запросов. Таким образом, запрос *statue of liberty* становится запросом *statue AND of AND liberty*, где любой документ,

содержащий все три слова («статуя», «из», «свобода»), будет соответствовать, предполагая оператор запроса по умолчанию AND.

Это булево сопоставление само по себе не дает отличных результатов, но в сочетании с ранжированием BM25 (обсуждаемым в разделе 3.2.1) оно может дать отличные результаты для наивного алгоритма, который не подразумевает истинного понимания терминов в домене.

В отличие от этого булевого анализа, также возможно преобразовать весь запрос в числовой векторный эмбединг, как обсуждалось в разделе 3.1.1. Мы рассмотрим поиск по плотным векторам с использованием LLM и эмбедингов позже в главах 13-14. Одним из преимуществ использования LLM и интерпретации запросов на основе эмбедингов является то, что эти методы обеспечивают лучшее представление запроса как единой смысловой единицы. Однако при использовании этого подхода логическая структура запроса иногда может быть потеряна, поэтому он может не работать хорошо для сценариев, где ваша булева логика должна быть сохранена или вы должны гарантировать, что в результатах поиска появятся определенные ключевые слова.

Последний способ анализа запроса – извлечение известных терминов и фраз из графа знаний. Мы использовали этот подход в нашем сквозном примере в разделе 7.3. Одним из преимуществ этого подхода является то, что в дополнение к предоставлению детального контроля над известным словарем он также позволяет явно моделировать интерпретацию определенных фраз и триггерных слов (*top*, *in*, *near*), чтобы отразить их функциональное значение, а не просто сопоставление ключевых слов. Недостатком этого подхода является то, что любые термины или фразы, *не существующие* в графе знаний, не могут быть так же легко извлечены и интерпретированы.

Поскольку мы углубимся в LLM в последующих главах, мы сосредоточимся в этой главе на явном анализе запросов с использованием графа знаний, поскольку явный анализ обеспечивает значительную настройку для домена, его дешево реализовать, и он позволяет нам включать все другие методы ИИ, которые мы уже изучили.

Реализация семантического парсера запросов

Первым шагом при семантической интерпретации запроса является определение терминов и фраз в запросе (фаза *парсинга*). В главе 6 мы рассмотрели, как выделить важные домен-специфичные термины и фразы из нашего контента и сигналов поведения пользователей. Их можно использовать в качестве списка известных сущностей для поддержки извлечения сущностей во входящих запросах.

Поскольку в списке известных фраз потенциально могут быть миллионы сущностей, эффективная структура, такая как *преобразователь конечного состояния* (FST), позволяет выполнять извлечение сущностей в этом масштабе всего за миллисекунды. Мы не будем здесь вдаваться в нюансы работы FST, но они обеспечивают очень компактное сжатие многих последовательностей терминов и очень быстрый поиск по этим последовательностям терминов, что позволяет молниеносно извлекать сущности.

В нашем примере поисковой системы Apache Solr реализован обработчик запросов *текстового теггера*¹, который специально создан для быстрого извлечения сущностей. Он позволяет вам индексировать любое количество терминов в индекс поиска, поэтому вы можете встроить этот индекс в FST и извлекать термины из этого индекса в любом входящем потоке текста.

В главе 6 мы создали списки домен-специфичных фраз, которые также включали альтернативные варианты написания. Мы можем сопоставить все эти термины в специально настроенную коллекцию сущностей вместе с любыми вариантами написания, чтобы обеспечить бесперебойное извлечение сущностей из входящих запросов. В следующем листинге рассматривается несколько типов данных сущностей в нашей коллекции сущностей.

Листинг 7.3 Данные сущностей, используемые для тегирования и извлечения

```
entities_dataframe = from_csv("data/reviews/entities.csv", log=False)
display_entities(entities_dataframe, limit=20)
```

Вывод:

Entities

id	surface_form	canonical_form	type	popularity
1	near	{location_distance}	semantic_function	90
2	in	{location_distance}	semantic_function	100
3	by	{location_distance}	semantic_function	90
4	by	{text_within_one_...}	semantic_function	10
5	near	{text_distance}	semantic_function	10
6	popular	{popular}	semantic_function	100
7	top	{popular}	semantic_function	100
8	best	{popular}	semantic_function	100
9	good	{popular}	semantic_function	100
10	violet	violet	color	100
11	violet crowne	violet crowne	brand	100
12	violet crowne cha...	violet crowne cha...	movie_theater	100
13	violet crown	violet crowne	brand	100
14	violet crown char...	violet crowne cha...	movie_theater	100
15	haystack	haystack conference	event	100
16	haystack conf	haystack conference	event	100
17	haystack conference	haystack conference	event	100
18	heystack	haystack conference	event	100
19	heystack conf	haystack conference	event	100
20	heystack conference	haystack conference	event	100

only showing top 20 rows

¹ Текстовый теггер в программировании – это инструмент, который позволяет идентифицировать фрагменты текста и присваивать им определенный тег при соответствии определенному условию. – *Прим. ред.*

... Entities continued

id	semantic_function
1	location_distance(query, position)
2	location_distance(query, position)
3	location_distance(query, position)
4	text_within_one_edit_distance(query, position)
5	text_distance(query, position)
6	popularity(query, position)
7	popularity(query, position)
8	popularity(query, position)
9	popularity(query, position)

Поля, представленные в таблице для сущностей в листинге 7.3, включают:

- `surface_form` – конкретный текст любого варианта написания, который мы хотим сопоставить в будущих запросах;
- `canonical_form` – «официальную» версию любого термина, который может иметь потенциально несколько поверхностных форм¹;
- `type` – классификацию (категорию) для термина в нашей области;
- `popular` – используется для приоритизации² различных значений одной и той же формы поверхности;
- `semantic_function` – присутствует только для сущностей типа «`semantic_function`». Используется для внедрения программной обработки специальных комбинаций ключевых слов.

В большинстве случаев `surface_form` и `canonical_form` будут одинаковыми, но наш экстрактор сущностей всегда будет находить соответствие `surface_form` и сопоставлять его с `canonical_form`, поэтому этот механизм используется для сопоставления нескольких вариантов написания сущности с одной официальной или «канонической» версией. Это можно использовать для обработки опечаток («`amin`» ⇒ «`admin`»), аббревиатур и инициалов («`cto`» ⇒ «`chief technology Officer`»), неоднозначных терминов («`cto`» ⇒ «`chief technology Officer`» по сравнению с «`cto`» ⇒ «`cancelled-to-order`») или даже сопоставления терминов с определенной логикой интерпретации (семантические функции), например «`near`» ⇒ {`location_distance`}.

¹ Поверхностная форма (англ. *surface form*) состоит из слова или группы слов, которые соответствуют лексическим единицам, например Парижу или Нью-Йорку. Поверхностные формы используются для выбора возможных записей в базе знаний и помогают разрешить неоднозначность через меры сходства. – Прим. ред.

² Приоритизация в программировании реализуется с помощью очереди с приоритетом (*priority queue*) – абстрактного типа данных, поддерживающего две обязательные операции – добавить элемент и извлечь максимум (минимум). Предполагается, что для каждого элемента можно вычислить его приоритет – действительное число или в общем случае элемент линейно упорядоченного множества. – Прим. ред.

Тип «semantic_function» – это особый тип, который мы рассмотрим в разделе 7.4.2; он позволяет использовать нелинейные условные правила синтаксического анализа запросов. Например, «если за словом near следует сущность, имеющая *географическую локацию* интерпретировать этот раздел запроса как географический фильтр».

В случае неоднозначного термина будет существовать несколько записей, содержащих одну и ту же поверхностную форму, сопоставленную с разными каноническими формами. В этом случае поле popularity будет указывать относительное значение, указывающее, какое значение является более распространенным (чем выше, тем популярнее).

Этот формат также является расширяемым – вы можете добавить поле vector, представляющее семантическое значение канонической формы, или поле related_terms, которое содержит другие термины со схожими значениями. Это позволит кешировать статическое представление значения canonical_form, что может быть намного эффективнее во время запроса, чем обращение к внешним моделям или графам знаний для известных терминов при каждом запросе.

Вызов экстрактора сущностей

В дополнение к коллекции reviews, созданной в листинге 7.1, нам также необходимо создать коллекцию entities, содержащую известные сущности для извлечения. Эта коллекция будет служить явным графом знаний, содержащим все сущности из листинга 7.3, а также список всех крупных городов мира. Следующий листинг настраивает и заполняет коллекцию сущностей.

Листинг 7.4. Создание коллекции сущностей

```
entities_collection = engine.create_collection("entities")
entities_dataframe = from_csv("data/reviews/entities.csv")
cities_dataframe = cities.load_dataframe("data/reviews/cities.csv")
entities_collection.write(entities_dataframe)
entities_collection.write(cities_dataframe)
```

Явные сущности и сущности городов индексируются в коллекции сущностей, которые будут использоваться для извлечения сущностей.

Создает коллекцию сущностей и настраивает ее для хранения нашего явного графа знаний сущностей для извлечения из запросов.

Вывод:

```
Wiping "entities" collection
Creating "entities" collection
Status: Success
Loading data/reviews/entities.csv
Schema:
root
  |-- id: integer (nullable = true)
  |-- surface_form: string (nullable = true)
  |-- canonical_form: string (nullable = true)
  |-- type: string (nullable = true)
```



```
|-- popularity: integer (nullable = true)
|-- semantic_function: string (nullable = true)
```

Loading Geonames...

Successfully written 21 documents

Successfully written 137581 documents

Один из моментов конфигурации, который мы должны выделить, – это настройка извлечения сущностей, которая происходит в `engine.create_collection("entities")`. В случае по умолчанию, когда Solr используется для обслуживания явного графа знаний для извлечения сущностей из запросов, функциональность текстового теггера Solr включается путем внутреннего внесения следующих изменений в конфигурацию:

- добавление конечной точки `/entities/tag` с помощью `TaggerRequestHandler` в Solr. Мы можем передавать запросы в эту конечную точку для выполнения извлечения сущностей любых сущностей, найденных в коллекции `entities`;
- добавление типа поля `tags` в схему, настроенную на использование FST в памяти, что позволяет выполнять компактное и быстрое тегирование из коллекции потенциально миллионов сущностей за миллисекунды;
- добавление поля `name_tag`, в которое копируется поле `surface_form`. Поле `name_tag` является типом поля `tags` и используется конечной точкой `/entities/tag` для сопоставления сущностей из запроса.

Если ваша поисковая система имеет собственные возможности по тегированию текста, конфигурация будет отличаться, но в следующем листинге показан код, соответствующий этим изменениям для реализации текстового теггера по умолчанию с использованием Apache Solr.

Листинг 7.5. Настройка текстового теггера Solr для извлечения сущностей

```
add_tag_type_commands = [{
  "add-field-type": {
    "name": "tag",
    "class": "solr.TextField",
    "postingsFormat": "FST50",
    "omitNorms": "true",
    "omitTermFreqAndPositions": "true",
    "indexAnalyzer": {
      "tokenizer": {"class": "solr.StandardTokenizerFactory"},
      "filters": [
        {"class": "solr.EnglishPossessiveFilterFactory"},
        {"class": "solr.ASCIIFoldingFilterFactory"},
        {"class": "solr.LowerCaseFilterFactory"},
        {"class": "solr.ConcatenateGraphFilterFactory",
          "preservePositionIncrements": "false"}]]},
  "name_tag": {
    "type": "tag",
    "copy-to": ["name_tag"]
  }
}]
```

Тип поля тега настраивается с использованием формата индекса Lucene FST50, что позволяет быстро сопоставлять данные в памяти.

ConcatenateGraphFilter – это специальный фильтр, используемый теггером текста для облегчения извлечения сущностей.

```

"queryAnalyzer": {
  "tokenizer": {"class": "solr.StandardTokenizerFactory"},
  "filters": [{"class": "solr.EnglishPossessiveFilterFactory"},
               {"class": "solr.ASCIIFoldingFilterFactory"},
               {"class": "solr.LowerCaseFilterFactory"}]}
{"add-field": {"name": "name_tag", "type": "tag",
               "stored": "false"}},
{"add-copy-field": {"source": "surface_form",
                    "dest": ["name_tag"]}}}

add_tag_request_handler_config = {
  "add-requesthandler": {
    "name": "/tag",
    "class": "solr.TaggerRequestHandler",
    "defaults": {
      "field": "name_tag",
      "json.nl": "map",
      "sort": "popularity desc",
      "matchText": "true",
      "fl": "id,surface_form,canonical_form,type,semantic_function,
            popularity,country,admin_area,*_p"
    }
  }
}

```

Мы добавляем поле `name_tag`, которое будем использовать для тегирования запросов по индексу.

Поле `name_tag` заполняется с использованием значения `surface_form`.

Обработчик запросов `/tag` настроен на использование значений, индексированных в поле `name_tag`, в качестве сущностей для извлечения из входящих запросов.

Если совпадает несколько сущностей (многозначность), возвращается самая популярная по умолчанию.

С созданной коллекцией `entities`, настроенным текстовым теггером и всеми проиндексированными сущностями мы теперь готовы выполнить извлечение сущностей по запросу. В следующем листинге мы запускаем запрос для `top kimchi near charlotte`.

Листинг 7.6. Извлечение сущностей для данного запроса

```

query = "top kimchi near charlotte"
entities_collection = engine.get_collection("entities")
extractor = get_entity_extractor(entities_collection)
query_entities = extractor.extract_entities(query)
print(query_entities)

```

Вывод:

```

{"query": "top kimchi near charlotte",
 "tags": [
   {"startOffset": 0, "endOffset": 3, "matchText": "top", "ids":
["7"]},
   {"startOffset": 11, "endOffset": 15, "matchText": "near",
"ids": ["1", "5"]},
   {"startOffset": 16, "endOffset": 25, "matchText": "charlotte",
"ids": ["4460243", "4612828", "4680560", "4988584", "5234793"]}],
 "entities": [
   {"id": "1", "surface_form": "near", "canonical_form": "{location_distance}",
"type": "semantic_function", "popularity": 90,
"semantic_function": "location_distance(query, position)"},
   {"id": "5", "surface_form": "near", "canonical_form": "{text_distance}",
"type": "semantic_function", "popularity": 10,

```

```

    "semantic_function": "text_distance(query, position)"},
    {"id": "7", "surface_form": "top", "canonical_form": "{popular}",
     "type": "semantic_function", "popularity": 100,
     "semantic_function": "popularity(query, position)"},
    {"id": "4460243", "canonical_form": "Charlotte", "surface_form": "Charlotte",
     "admin_area": "NC", "popularity": 827097, "type": "city",
     "location_coordinates": "35.22709, -80.84313"},
    {"id": "4612828", "canonical_form": "Charlotte", "surface_form": "Charlotte",
     "admin_area": "TN", "popularity": 1506, "type": "city",
     "location_coordinates": "36.17728, -87.33973"},
    {"id": "4680560", "canonical_form": "Charlotte", "surface_form": "Charlotte",
     "admin_area": "TX", "popularity": 1815, "type": "city",
     "location_coordinates": "28.86192, -98.70641"},
    {"id": "4988584", "canonical_form": "Charlotte", "surface_form": "Charlotte",
     "admin_area": "MI", "popularity": 9054, "type": "city",
     "location_coordinates": "42.56365, -84.83582"},
    {"id": "5234793", "canonical_form": "Charlotte", "surface_form": "Charlotte",
     "admin_area": "VT", "popularity": 3861, "type": "city",
     "location_coordinates": "44.30977, -73.26096"}]]

```

Ответ включает три ключевых раздела:

- `query` – запрос, который был тегирован;
- `tags` – список текстовых фраз, найденных во входящем запросе, вместе со смещениями символов в тексте (начальная и конечная позиции) и список всех возможных совпадений сущностей (канонические формы) для каждого тега (поверхностная форма);
- `entities` – список идентификаторов документов для сопоставления сущностей, которые могут соответствовать одному из сопоставленных тегов.

Ранее мы описывали неоднозначные термины, где одна поверхностная форма может соответствовать нескольким каноническим формам. В нашем примере первый тег – `{'startOffset': 0, 'endOffset': 3, 'matchText': 'top', 'ids': ['7']}`. Это означает, что текст «top» был сопоставлен, начиная с позиции 0 и заканчивая позицией 3 во входном `top kinchi near charlotte`. Он также перечисляет только одну запись в `ids`, что означает, что существует только одно возможное значение (каноническое представление). Однако для двух других тегов перечислено несколько `ids`, что делает их неоднозначными тегами:

- `{"startOffset": 11, "endOffset": 15, "matchText": "near", "ids": ["1", "5"]}`
- `{"startOffset": 16, "endOffset": 25, "matchText": "charlotte", "ids": ["4460243", "4612828", "4680560", "4988584", "5234793"]}`

Это означает, что было две канонические формы (перечислены два `ids`) для поверхностной формы «near» и пять канонических форм для поверхностной формы «charlotte». В разделе `entities` мы также можем увидеть все различные записи сущностей, связанные со списками идентификаторов в тегах.

В этой главе мы будем упрощать все, всегда используя каноническую форму с самой высокой `popularity`. Для городов мы указали население

города в поле popularity, что означает, что выбранный «charlotte» – это Шарлотт, Северная Каролина, США (Шарлотт с самым большим населением в мире). Для наших других сущностей популярность была указана вручную в entities.csv из листинга 7.3. Вы также можете указать популярность, используя значение бустинга сигналов (если вы вывели свои сущности из сигналов, что будет подробно рассмотрено в главе 8) или используя количество документов, содержащих сущность в вашем индексе, в качестве прокси для популярности.

Вы также можете найти полезным использовать контекст, специфичный для пользователя, или контекст, специфичный для запроса, чтобы выбрать наиболее подходящую сущность. Например, если вы устраняете неоднозначность локаций, можете повысить популярность с помощью расчета географического расстояния, чтобы локация, расположенные ближе к пользователю, получили более высокий вес. Если сущность является ключевой фразой, вы можете альтернативно использовать SKG для классификации запроса или загрузки вектора термина и повышения канонической формы, которая является лучшим понятийным соответствием для всего запроса.

С нашими query_entities, доступными из графа знаний, мы теперь можем сгенерировать удобную для пользователя версию исходного запроса с идентифицированными тегированными сущностями. Следующий листинг реализует эту функцию generate_tagged_query.

Листинг 7.7. Генерация тегированного запроса

```
def generate_tagged_query(extracted_entities):
    query = extracted_entities["query"]
    last_end = 0
    tagged_query = ""
    for tag in extracted_entities["tags"]:
        next_text = query[last_end:tag["startOffset"]].strip()
        if len(next_text) > 0:
            tagged_query += " " + next_text
            tagged_query += " {" + tag["matchText"] + "}"
            last_end = tag["endOffset"]
    if last_end < len(query):
        final_text = query[last_end:len(query)].strip()
        if len(final_text):
            tagged_query += " " + final_text
    return tagged_query
```

Реконструирует запрос с тегированными сущностями.

Заключает известные сущности в фигурные скобки, чтобы отделить их от обычных ключевых слов.

```
tagged_query = generate_tagged_query(query_entities)
print(tagged_query)
```

Вывод:

```
{top} kimchi {near} {charlotte}
```

Из этого тегированного запроса мы теперь можем видеть, что ключевые слова «top», «near» и «charlotte» сопоставляются с известными

сущностями, в то время как «kimchi» является неизвестным ключевым словом. Этот формат является полезным и удобным для пользователя представлением запроса, но он слишком прост для представления метаданных, связанных с каждой сущностью. Поскольку нам необходимо программно обрабатывать сущности и их семантические взаимодействия для обогащения запроса, мы также реализуем более структурированное представление семантически парсированного запроса, которое мы назовем `query_tree`.

Вместо чистого текстового запроса этот `query_tree` представляет собой структуру строго типизированных узлов в запросе, представленном как объекты JSON. Листинг 7.8 демонстрирует функцию `generate_query_tree`, которая возвращает дерево запроса из входящих данных извлечения сущности (`query_entities`).

Листинг 7.8. Генерация типизированного дерева запросов из пользовательского запроса

```
def generate_query_tree(extracted_entities):
    query = extracted_entities["query"]
    entities = {entity["id"]: entity for entity
                in extracted_entities["entities"]}

    query_tree = []
    last_end = 0

    for tag in extracted_entities["tags"]:
        best_entity = entities[tag["ids"][0]]
        for entity_id in tag["ids"]:
            if (entities[entity_id]["popularity"] >
                best_entity["popularity"]):
                best_entity = entities[entity_id]

        next_text = query[last_end:tag["startOffset"]].strip()
        if next_text:
            query_tree.append({"type": "keyword",
                              "surface_form": next_text,
                              "canonical_form": next_text})

        query_tree.append(best_entity)
        last_end = tag["endOffset"]

    if last_end < len(query):
        final_text = query[last_end:len(query)].strip()
        if final_text:
            query_tree.append({"type": "keyword",
                              "surface_form": final_text,
                              "canonical_form": final_text})

    return query_tree

parsed_query = generate_query_tree(query_entities)
display(parsed_query)
```

Делает маппинг ID сущностей на сущности.

Выбирает наиболее популярную каноническую форму для сущности по умолчанию.

Добавляет объект сущности в дерево запроса в соответствующей позиции в запросе.

Назначает тип ключевого слова любому нетегированному тексту в качестве резервного варианта.

Любой текст после последней отмеченной сущности также будет считаться ключевым словом.

Вывод:

```
[{"semantic_function": "popularity(query, position)", "popularity": 100,
  "id": "7", "surface_form": "top", "type": "semantic_function",
  "canonical_form": "{popular}"},
 {"type": "keyword", "surface_form": "kimchi", "canonical_form":
"kimchi"},
 {"semantic_function": "location_distance(query, position)",
"popularity": 90,
  "id": "1", "surface_form": "near", "type": "semantic_function",
  "canonical_form": "{location_distance}"},
 {"country": "US", "admin_area": "NC", "popularity": 827097,
  "id": "4460243", "surface_form": "Charlotte", "type": "city",
  "location_coordinates": "35.22709,-80.84313",
  "canonical_form": "Charlotte"}]
```

Теперь у нас есть несколько представлений запроса и тегированных сущностей:

- `tagger_data` – вывод листинга 7.6;
- `tagged_query` – вывод листинга 7.7;
- `parsed_query` – вывод листинга 7.8.

Вывод `parsed_query` – это сериализация базового объекта `query_tree`, полностью представляющая все ключевые слова и сущности вместе с их метаданными. На этом этапе начальная фаза *парсинга* (сбора и анализа данных с помощью скриптов), которая преобразует запрос в типизированные сущности, завершена, и мы можем начать использовать отношения между сущностями для лучшего обогащения запроса.

7.4.2. Обогащение запроса для семантического поиска

Фаза *обогащения* нашего конвейера интерпретации запроса фокусируется на понимании взаимосвязей между сущностями в запросе и на том, как лучше всего их интерпретировать и представлять для оптимальной релевантности результатов поиска.

Большая часть этой книги уже была и будет фокусироваться на фазе обогащения. Глава 4 представила краудсорсинговую релевантность, которая является способом обогащения определенных ключевых фраз информацией о том, какие документы являются наиболее релевантными, на основе предыдущих взаимодействий пользователей. Глава 5 фокусировалась на графах знаний, которые предоставляют способ обогащения определенных ключевых фраз тематическими классификациями и поиска других терминов, которые тесно связаны. В главе 6 мы реализовали алгоритмы для поиска синонимов, орфографических ошибок и связанных терминов, которые можно использовать для обогащения запросов путем дополнения или замены проанализированных терминов лучшими, изученными версиями. В следующих главах по бустингу сигналов, персонализации и плотному векторному поиску по эмбедингам также будут представлены новые способы интерпретации проанализированных сущностей и обогащения запросов для оптимизации релевантности.

Все эти методы – инструменты в вашем инструментальном наборе, но лучший способ объединить их для любой конкретной реализации будет домен-специфичным, поэтому в наших примерах мы избежим чрезмерного обобщения. Вместо этого сосредоточимся на простой сквозной реализации, которая связала бы все вместе таким образом, что можно было бы легко подключить другие модели. Наша простая реализация будет состоять из двух компонентов:

- реализации семантической функции, которая позволяет вводить динамические и нелинейные семантические правила для каждого домена;
- SKG для поиска связанных терминов для неизвестных ключевых слов и классификаций запросов.

У вас уже есть инструменты расширения структуры анализа запросов для обработки других типов обогащения из предыдущих глав. Например, вы можете использовать отображения из поверхностной формы в каноническую форму для обработки всех альтернативных представлений, изученных в главе 6. Аналогично, добавляя дополнительные поля к каждой сущности в коллекции `entities`, вы можете вводить бустинг сигналов, связанные термины, классификации запросов или векторы, делая их доступными для использования, как только запросы будут проанализированы. Давайте начнем нашу реализацию обогащения с обсуждения семантических функций.

Реализация семантических функций

Семантическая функция – это нелинейная функция, которую можно применять во время анализа запроса и обогащения для лучшей интерпретации значения окружающих терминов. Наш предыдущий пример `top kimchi near charlotte` содержит два термина, которые сопоставляются с семантическими функциями: «top» и «near». Термин «top» имеет очень специфичное для домена значение: отдавать приоритет документам с наивысшим рейтингом (количество звезд в обзоре). Аналогично термин «near» не является ключевым словом, которое должно сопоставляться; вместо этого он изменяет значение последующих терминов, пытаясь упорядочить их в географической локации. Из листинга 7.3 вы увидите следующие сущности, ссылающиеся на семантические функции:

Semantic Function Entities

surface	canonical_form	popularity	semantic_function
near	{location_distance}	90	location_distance(query, position)
in	{location_distance}	100	location_distance(query, position)
by	{location_distance}	90	location_distance(query, position)
by	{text_within_on...}	10	text_within_one_edit_distance(...)
near	{text_distance}	10	text_distance(query, position)
popular	{popular}	100	popularity(query, position)
top	{popular}	100	popularity(query, position)
best	{popular}	100	popularity(query, position)
good	{popular}	100	popularity(query, position)

Вы заметите, что формы поверхности «top», «popular», «good» и «best» все преобразуются в каноническую форму {popularity}, которая представлена семантической функцией `popular(query, position)` в следующем листинге.

Листинг 7.9. Семантическая функция, которая учитывает популярность

```
def popularity(query, position):
    if len(query["query_tree"]) - 1 > position:
        query["query_tree"][position] = {
            "type": "transformed",
            "syntax": "solr",
            "query": ' +{!func v="mul(if(stars_rating,stars_rating,0),20)"}'
        }
    return True
return False
```

Другой узел дерева запросов должен следовать за узлом популярности («высочайшая гора», а не «вершина горы»).

Заменяет узел {popularity} в дереве запросов новым узлом, который представляет повышение релевантности популярности.

Возвращает, семантическая функция сработала. Если False, то можно попытаться выполнить другую перекрывающуюся функцию с более низким приоритетом.

Эта функция популярности позволяет нам применять логику семантической интерпретации для манипулирования деревом запроса. Если бы дерево запроса заканчивалось ключевым словом «top», функция вернула бы False, и никакие корректировки не были бы сделаны. Аналогично, если бы другой функции был назначен более высокий приоритет (как указано в коллекции `entities`), она могла бы удалить сущность {popularity} еще до того, как ее функция была бы выполнена.

Функция `location_distance` немного сложнее, как показано в следующем листинге.

Листинг 7.10. Семантическая функция, которая учитывает локацию

```
def location_distance(query, position):
    if len(query["query_tree"]) - 1 > position:
        next_entity = query["query_tree"][position + 1]
        if next_entity["type"] == "city":
            query["query_tree"].pop(position + 1)
            query["query_tree"][position] = {
                "type": "transformed",
                "syntax": "solr",
                "query": create_geo_filter(
                    next_entity['location_coordinates'],
                    "location_coordinates", 50)
            }
        return True
    return False
```

Добавляет заменяющую сущность с фильтром радиуса.

Функция должна изменить следующую сущность, чтобы выполнение было успешным.

Следующая сущность должна иметь тип локации (город).

Удаляет следующую сущность, так как это локация, которая теперь будет заменена фильтром на основе радиуса.

Если следующая сущность не была городом, функция не применяется.

```
def create_geo_filter(coordinates, field, distance_KM):
    return f'+{!geofilt d={distance_KM} sfield="{field}" pt="{coordinates}"}'
```

Как вы можете видеть, наша реализация семантических функций позволяет условно применять любую произвольную логику при интерпретации запросов. Если хотите, вы даже можете вызывать внешние графы знаний или другие источники данных, чтобы получить дополнительную информацию для лучшей интерпретации запроса.

Вы могли заметить, что все поверхностные формы «near», «in» и «by» сопоставляются с канонической формой {location_distance}, которая представлена функцией `location_distance(query, position)`. Эта функция хорошо работает, если за одним из этих терминов следует локация, но что, если кто-то ищет *chief near officer*? В этом контексте конечный пользователь мог иметь в виду «найти термин *chief* близко к термину *officer* в документе» – по сути, поиск по расстоянию редактирования. Обратите внимание, что также есть сопоставление сущностей «near» $\Rightarrow$ {text_distance}, которое можно вызывать условно для этого резервного варианта использования, если семантическая функция сущности {location_distance} возвращает False.

Семантические функции могут быть реализованы многими различными способами, но наш пример реализации обеспечивает высоконастраиваемый способ кодирования динамических семантических шаблонов в конвейере интерпретации запросов для наилучшей связи с различными подходами поиска на основе ИИ, доступными для вашего поискового приложения. Мы показываем эту реализацию в функции `process_semantic_functions` в следующем листинге, которая проходит по дереву запроса для вызова всех соответствующих семантических функций.

Листинг 7.11. Обработка всех семантических функций в дереве запроса

```
def process_semantic_functions(query_tree):
    position = 0
    while position < len(query_tree):
        node = query_tree[position]
        if node["type"] == "semantic_function":
            query = {"query_tree": query_tree}
            command_successful = eval(node["semantic_function"])
            if not command_successful:
                node["type"] = "invalid_semantic_function"
            position += 1
    return query_tree
```

Проходит по всем предметам в дереве запроса, ищет семантические функции для выполнения.

Переменные запроса и позиции передаются в семантическую функцию eval.

Обновляет тип любых неудачных семантических функций.

Динамически оценивает семантические функции, которые дополняют дерево запроса.

Поскольку семантические функции хранятся как часть их сущности из коллекции `entities`, мы выполняем позднее связывание¹ этих функ-

¹ Функция связывания – это процесс привязки функции к определенному контексту. – Прим. ред.

ций (используя функцию `eval` Python). Это позволяет вам в любое время подключать новые семантические функции в коллекцию `entities` без необходимости изменения кода приложения.

Поскольку семантические функции могут быть успешными или нет в зависимости от окружающих узлов контекста, каждая семантическая функция должна возвращать `True` или `False`, чтобы позволить логике обработки определить, как действовать с остальной частью дерева запроса.

Интеграция SKG

В этом разделе мы интегрируем SKG (обсуждаемый в главе 5) в наш процесс обогащения запроса.

Ваша коллекция `entities`, вероятно, будет содержать много изученных сущностей с использованием методов из главы 6. Вы также можете использовать SKG или другой метод для классификации известных сущностей или для создания списков связанных терминов. Если вы это сделаете, мы рекомендуем добавлять классификации и связанные термины в качестве дополнительных полей в коллекцию `entities` для кеширования ответов для более быстрого поиска во время запроса.

Для нашей реализации мы вызовем SKG в реальном времени для обогащения *неизвестных* терминов. Этот подход вводит связанные ключевые слова для всех неизвестных ключевых фраз в запросе, что может генерировать много ложных срабатываний. Вероятно, вы хотите быть более консервативным в любой производственной реализации, но реализация этого полезна для обучения и экспериментов. Следующий листинг демонстрирует, как искать ключевые фразы и просматривать нашу коллекцию отзывов как SKG.

Листинг 7.12. Получение связанных терминов и категорий из SKG

```
def get_enrichments(collection, keyword, limit=4):
    enrichments = {}
    nodes_to_traverse = [{"field": "content",
                          "values": [keyword],
                          "default_operator": "OR"},
                        [{"name": "related_terms",
                          "field": "content",
                          "limit": limit},
                        {"name": "doc_type",
                          "field": "doc_type",
                          "limit": 1}]]

    skg = get_semantic_knowledge_graph(collection)
    traversals = skg.traverse(*nodes_to_traverse)
    if "traversals" not in traversals["graph"][0]["values"][keyword]:
        return enrichments

    nested_traversals = traversals["graph"][0]["values"] \
        [keyword]["traversals"]
```

Возвращает 4 топовых связанных термина для ключевого слова.

Начальным узлом для обхода SKG является запрос для переданного ключевого слова в поле контента.

Возвращает 1 топовый doc_type (категорию) для ключевого слова.

Возвращает пустое значение, если обогащения не найдены.

```

doc_types = list(filter(lambda t: t["name"] == "doc_type",
                        nested_traversals))
if doc_types:
    enrichments["category"] = next(iter(doc_types[0]["values"]))
    related_terms = list(filter(lambda t: t["name"] == "related_terms",
                                nested_traversals))
if related_terms:
    term_vector = ""
    for term, data in related_terms[0]["values"].items():
        term_vector += f'{term}^{round(data["relatedness"], 4)} '
    enrichments["term_vector"] = term_vector.strip()

return enrichments

query = "kimchi"
get_enrichments(reviews_collection, query)

```

Возвращает обнаруженные категории из обхода.

Создает усиленный запрос из обнаруженных связанных терминов, усиленных их родством.

Получает обогащения для ключевого слова «кимчи».

Вывод листинга 7.12 для ключевого слова «kimchi» выглядит следующим образом:

```

{"category": "Korean",
 "term_vector": "kimchi^0.9193 korean^0.7069 banchan^0.6593
 bulgogi^0.5497"}

```

Вот некоторые примеры выводов SKG для других потенциальных ключевых слов:

- *bbq*:


```

{"category": "Barbeque",
 "term_vector": "bbq^0.9191 ribs^0.6187 pork^0.5992 brisket^0.5691"}

```
- *korean bbq*:


```

{"category": "Korean",
 "term_vector": "korean^0.7754 bbq^0.6716 banchan^0.5534 sariwon^0.5211"}

```
- *lasagna*:


```

{"category": "Italian",
 "term_vector": "lasagna^0.9193 alfredo^0.3992 pasta^0.3909
 ➡italian^0.3742"}

```
- *karaoke*:


```

{"category": "Karaoke",
 "term_vector": "karaoke^0.9193 sing^0.6423 songs^0.5256 song^0.4118"}

```
- *drive through*:


```

{"category": "Fast Food",
 "term_vector": "drive^0.7428 through^0.6331 mcdonald's^0.2873
 ➡window^0.2643"}

```

Чтобы завершить нашу фазу *обогащения*, нам нужно применить функцию `get_enrichments` и ранее обсуждавшуюся функцию `process_semantic_functions` к нашему дереву запросов.

Листинг 7.13. Обогащение узлов дерева запросов

```
def enrich(collection, query_tree):
    query_tree = process_semantic_functions(query_tree)
    for item in query_tree:
        if item["type"] == "keyword":
            enrichments = get_enrichments(collection, item["surface_form"])
            if enrichments:
                item["type"] = "skg_enriched"
                item["enrichments"] = enrichments
    return query_tree
```

Берет все неизвестные ключевые фразы и ищет их в SKG.

Циркулирует по дереву запросов и обрабатывает все семантические функции.

Если найдены обогащения, применяет их к узлу.

Эта функция `enrich` охватывает всю фазу обогащения, обрабатывая все семантические функции, а затем обогащая все оставшиеся неизвестные ключевые слова с помощью SKG. Однако, прежде чем перейти к фазе преобразования, давайте быстро рассмотрим альтернативный подход к расширению ключевых слов на основе SKG, который мы реализовали.

7.4.3. Разреженные лексические модели и модели расширения

До сих пор в этой книге мы рассмотрели два основных подхода к поиску: *лексический поиск* – сопоставление и ранжирование на основе определенных терминов или признаков в запросе – и *семантический поиск* – сопоставление и ранжирование на основе смысла запроса. Вы также познакомились с двумя основными подходами к представлению запросов: разреженные векторы (векторы с очень небольшим количеством ненулевых значений) и плотные векторы (векторы в основном с ненулевыми значениями). Лексический поиск ключевых слов обычно реализуется с использованием инвертированного индекса, который содержит разреженное векторное представление каждого документа с измерением для каждого термина в индексе. Семантический поиск также обычно реализуется с использованием плотного векторного представления для поиска по эмбедингам.

Сравнение поиска по разреженному вектору, по плотному вектору и лексического поиска

Из-за вычислительных затрат плотные векторные представления обычно имеют ограниченное количество измерений (сотни или тысячи), которые плотно сжимают семантическое представление данных, тогда как разреженные векторные представления могут легко иметь сотни тысяч или десятки миллионов измерений, которые представляют более идентифицируемые термины или атрибуты. Лексический поиск по ключевым словам обычно реализуется с использованием инвертированного индекса, который содержит разреженное векторное представление каждого документа с измерением для каждого термина в индексе. Семантический поиск обычно реализуется с использованием плотного векторного представления для поиска по эмбедингам.

Из-за этих тенденций многие ошибочно считают термин «семантический поиск» синонимом плотного векторного поиска на эмбедингах, но это отбрасывает богатую историю гораздо более объяснимых и гибких подходов к семантическому поиску на основе разреженных векторов и графов. В этой главе освещаются некоторые из этих подходов, а главы 13–15 более подробно рассматривают методы плотного векторного поиска.

Однако, как вы уже видели в этой главе, семантический поиск также может быть реализован с использованием разреженных векторов и в контексте типичных лексических запросов. Хотя мы реализовали семантический анализ запросов, который работает непосредственно с запросами пользователей, мы также сгенерировали расширения запросов с использованием SKG для генерации разреженных векторов терминов и весов для обеспечения семантического поиска.

Существуют и другие методы для такого рода расширения запросов, такие как SPLADE (Sparse Lexical and Expansion). Вместо использования инвертированного индекса в качестве языковой модели подход SPLADE (<https://arxiv.org/pdf/2107.05720>) использует готовую языковую модель для генерации контекстуализированных токенов. Мы не будем использовать SPLADE (или SPLADE V2, или последующие версии), поскольку он не был выпущен по лицензии, допускающей коммерческое использование, но листинг 7.14 демонстрирует пример выходных данных из альтернативной реализации с открытым исходным кодом (SPLADE++) для тех же примеров запросов, которые мы только что протестировали с подходом SKG в разделе 7.4.2.

Листинг 7.14. Расширение запросов с помощью SPLADE++

```
from spladerunner import Expander
expander = Expander('Splade_PP_en_v1', 128)
queries = ["kimchi", "bbq", "korean bbq",
           "lasagna", "karaoke", "drive through"]

for query in queries:
    sparse_vec = expander.expand(query,
                                outformat="lucene")[0]
    print(sparse_vec)
```

Указывает имя модели SPLADE++ и максимальную длину последовательности.

Генерирует разреженный лексический вектор.

Возвращает метки токенов (строки) вместо ID (целые числа).

Вот выходные данные расширения SPLADE++:

■ *kimchi*:

```
{ "kim": 3.11, "##chi": 3.04, "ki": 1.52, ",": 0.92, "who": 0.72,
  "brand": 0.56, "genre": 0.46, "chi": 0.45, "##chy": 0.45,
  "company": 0.41, "only": 0.39, "take": 0.31, "club": 0.25,
  "species": 0.22, "color": 0.16, "type": 0.15, "but": 0.13,
  "dish": 0.12, "hotel": 0.11, "music": 0.09, "style": 0.08,
  "name": 0.06, "religion": 0.01 }
```

- *bbq*:

```
{
  "bb": 2.78, "grill": 1.85, "barbecue": 1.36, "dinner": 0.91,
  "##q": 0.78, "dish": 0.77, "restaurant": 0.65, "sport": 0.46,
  "food": 0.34, "style": 0.34, "eat": 0.24, "a": 0.23, "genre": 0.12,
  "definition": 0.09}

```
- *korean bbq*:

```
{
  "korean": 2.84, "korea": 2.56, "bb": 2.23, "grill": 1.58, "dish": 1.21,
  "restaurant": 1.18, "barbecue": 0.79, "kim": 0.67, "food": 0.64,
  "dinner": 0.39, "restaurants": 0.32, "japanese": 0.31, "eat": 0.27,
  "hotel": 0.16, "famous": 0.11, "brand": 0.11, "##q": 0.06, "diner": 0.02}

```
- *lasagna*:

```
{
  "las": 2.87, "##ag": 2.85, "##na": 2.39, ",": 0.84, "she": 0.5,
  "species": 0.34, "hotel": 0.33, "club": 0.31, "location": 0.3,
  "festival": 0.29, "company": 0.27, "her": 0.2, "city": 0.12,
  "genre": 0.05}

```
- *karaoke*:

```
{
  "kara": 3.04, "##oke": 2.87, "music": 1.31, "lara": 1.07,
  "song": 1.03, "dance": 0.97, "style": 0.94, "sara": 0.81,
  "genre": 0.75, "dress": 0.48, "dish": 0.44, "singer": 0.37,
  "hannah": 0.36, "brand": 0.31, "who": 0.29, "culture": 0.21,
  "she": 0.17, "mix": 0.17, "popular": 0.12, "girl": 0.12,
  "kelly": 0.08, "wedding": 0.0}

```
- *drive through*:

```
{
  "through": 2.94, "drive": 2.87, "driving": 2.34, "past": 1.75,
  "drives": 1.65, "thru": 1.44, "driven": 1.22, "enter": 0.81,
  "drove": 0.81, "pierce": 0.75, "in": 0.72, "by": 0.71, "into": 0.64,
  "travel": 0.59, "mark": 0.51, ";": 0.44, "clear": 0.41,
  "transport": 0.41, "route": 0.39, "within": 0.36, "vehicle": 0.3,
  "via": 0.15}

```

Обратите внимание, что параметр `outputformat=lucene` приводит к возврату токенов (ключевых слов или частичных ключевых слов) вместо целочисленных ID токенов, поскольку просмотр токенов помогает нам лучше интерпретировать результаты.

При сравнении этого вывода с ранее показанным выводом SKG для тех же запросов вы можете заметить следующие различия:

- вывод SKG возвращает фактические термины в индексе, тогда как вывод в стиле SPLADE возвращает токены из LLM. Это означает, что вы можете использовать вывод SKG («lasagna», «alfredo», «pasta») напрямую для поиска по полям в ваших документах, тогда как токены SPLADE (las, ##ag, na##) необходимо будет сгенерировать из SPLADE для всех документов и проиндексировать, чтобы запрос в стиле SPLADE сопоставлялся с правильными токенами во время запроса;
- разреженные векторы SKG, как правило, выглядят более ясными и релевантными набору данных (обзоры ресторанов) для терминов в домене. Например, для запроса *bbq* SKG возвращает {"bbq": 0.9191, "ribs": 0.6186, "pork": 0.5991, "brisket": 0.569}, тогда

как SPLADE возвращает {'bb': 2.78, 'grill': 1.85, 'barbecue': 1.36, 'dinner': 0.91, '##q': 0.78, 'dish': 0.77, 'restaurant': 0.65, 'sport': 0.46, 'food': 0.34, ...}. Эта низкая производительность модели SPLADE, по сравнению с моделью SKG, в основном обусловлена тем, что SPLADE не обучается на данных в поисковом индексе, тогда как SKG использует данные в поисковом индексе напрямую в качестве своей языковой модели. Тонкая настройка модели на основе SPLADE поможет закрыть этот пробел;

- модель SKG более гибкая, поскольку она может возвращать отношения по нескольким измерениям. Обратите внимание в последнем разделе, что мы не только вернули разреженный вектор связанных терминов, но и классификацию запроса;
- обе модели, SPLADE и SKG, являются контекстно-зависимыми. SPLADE изначально взвешивает каждый токен на основе всего контекста кодируемого запроса (или документа), и запрос SKG может также (опционально) использовать любой переданный контекст для запроса или документа, чтобы контекстуализировать веса своих токенов. Модели на основе SPLADE, как правило, лучше всего подходят для более длинных известных контекстов (например, общих документов), тогда как модель SKG больше оптимизирована для более коротких, специфичных для домена контекстов (например, внутридоменных запросов), но они обе работают и представляют новые методы для разреженного векторного или лексически ориентированного семантического поиска.

Мы решили использовать подход на основе SKG вместо SPLADE в этой главе из-за его способности также классифицировать запросы и дополнительно контекстуализировать запросы для устранения неоднозначности смысла запроса, но аналогичные понятия применяются для реализации семантического поиска на основе разреженного вектора независимо от того, какую модель вы выберете, поэтому хорошо быть знакомым с несколькими методами.

В следующем разделе мы рассмотрим преобразование обогащенного дерева запросов в синтаксис запроса, специфичный для поисковой системы, для отправки в поисковый движок.

7.4.4. Преобразование запроса для семантического поиска

Теперь, когда запрос пользователя проанализирован и обогащен, пришло время преобразовать дерево запроса в соответствующий синтаксис, специфичный для поисковой системы.

Во время этой фазы *преобразования* мы вызываем адаптер для преобразования дерева запроса в наиболее полезное представление запроса, специфичное для поисковой системы – Solr в нашей реализации по умолчанию. В случае наших семантических функций (функции `popularity` и `location_distance`) мы уже внедрили этот синтаксис, специфичный для поисковой системы (`{"type": "transformed", "syntax": "solr"}`), непосредственно в обогащенные узлы в дереве запроса. Мы могли бы в качестве альтернативы немного абстрагировать это, создав общее

промежуточное представление вывода каждой семантической функции, а затем дождаться фазы преобразования для преобразования в синтаксис, специфичный для поисковой системы (Solr, OpenSearch и т. д.), но мы решили избегать промежуточных представлений, чтобы сделать примеры более простыми. Если вы запустите код с использованием другого движка (как объяснено в приложении В), вы увидите синтаксис для этого движка в преобразованных узлах.

В следующем листинге показана функция `transform_query`, которая берет обогащенное дерево запроса и преобразует каждый из его узлов в узлы, специфичные для поисковой системы.

Листинг 7.15. Преобразование дерева запроса в синтаксис, специфичный для поисковой системы

```
def transform_query(query_tree):
    for i, item in enumerate(query_tree):
        match item["type"]:
            case "transformed":
                continue
            case "skg_enriched":
                enrichments = item["enrichments"]
                if "term_vector" in enrichments:
                    query_string = enrichments["term_vector"]
                    if "category" in enrichments:
                        query_string += f' +doc_type:"{enrichments["category"]}"'
                    transformed_query =
                        ➔' +{!edismax v="' + escape_quotes(query_string) + '"}'

                else:
                    continue
            case "color":
                transformed_query = f'+colors:"{item["canonical_form"]}"'
            case "known_item" | "event":
                transformed_query = f'+name:"{item["canonical_form"]}"'
            case "city":
                transformed_query = f'+city:"{str(item["canonical_form"])}"'
            case "brand":
                transformed_query = f'+brand:"{item["canonical_form"]}"'
            case _:
                transformed_query = "+{!edismax v=\"\" +
                    ➔escape_quotes(item["surface_form"]) + "\""

        query_tree[i] = {"type": "transformed",
                        "syntax": "solr",
                        "query": transformed_query}

    return query_tree
```

Если элемент дерева запроса уже был преобразован в синтаксис, специфичный для поисковой системы, нет необходимости обрабатывать его дальше.

Создает обогащенный запрос для обогащенных узлов.

Обрабатывает преобразование других потенциальных элементов дерева запроса с помощью пользовательской логики обработки типов.

Обозначает каждый преобразованный узел дерева запроса с помощью синтаксиса и запроса, специфичных для поисковой системы.

Для всех других типов без пользовательской логики преобразования просто выполняет поиск по их поверхностной форме.

```
enriched_query_tree = enrich(reviews_collection, query_tree)
processed_query_tree = transform_query(enriched_query_tree)
display(processed_query_tree)
```

Вывод:

```
[{"type": "transformed",
  "syntax": "solr",
  "query": "+{!func v=\"mul(if(stars_rating,stars_rating,0),20)\"}",
  {"type": "transformed",
    "syntax": "solr",
    "query": "{!edismax v=\"kimchi^0.9193 korean^0.7069 banchan^0.6593
      ➔+doc_type:\\\"Korean\\\"\\\"\\\"}",
    {"type": "transformed",
      "syntax": "solr",
      "query": "+{!geofilt d=50 sfield=\"location_coordinates\"
        ➔pt=\"35.22709,-80.84313\\\"\\\"}]}
```

На этом этапе все узлы в дереве запроса были преобразованы в узлы {'type': 'transformed', 'syntax': engine}, что означает, что они внутри содержат синтаксис, специфичный для поисковой системы, необходимый для генерации окончательного запроса к настроенной поисковой системе. Теперь мы готовы преобразовать дерево запроса в строку и отправить запрос в поисковую систему.

7.4.5. Поиск с помощью семантически улучшенного запроса

Последний шаг нашего процесса семантического поиска – фаза *поиска*. Мы преобразуем полностью преобразованное `query_tree` в запрос, запускаем запрос в поисковой системе и возвращаем результаты конечному пользователю.

Листинг 7.16. Выполнение запроса

```
def to_query(query_tree):
    return [[node["query"] for node in query_tree]
transformed_query = to_query(query_tree)
reviews_collection = engine.get_collection("reviews")
reviews_collection.search(query=transformed_query)
```

Результаты поиска для нашего запроса `top kimchi near charlotte` вернут именно то, что было показано в нашем сквозном примере в разделе 7.3. Поскольку мы знаем, что теперь можем обрабатывать вариации семантических функций («in» или «near» для локаций; «good», или «popular», или «top» для популярности), мы покажем вывод для слегка измененного запроса: `good kimchi in charlotte`. Если вы сравните вывод для этого варианта, показанного на рис. 7.8, с выводом для исходного запроса `top kimchi near charlotte`, вы увидите, что они выдают точно такой же преобразованный запрос и окончательный набор результатов поиска, как и рис. 7.5 и 7.7 ранее в этой главе.

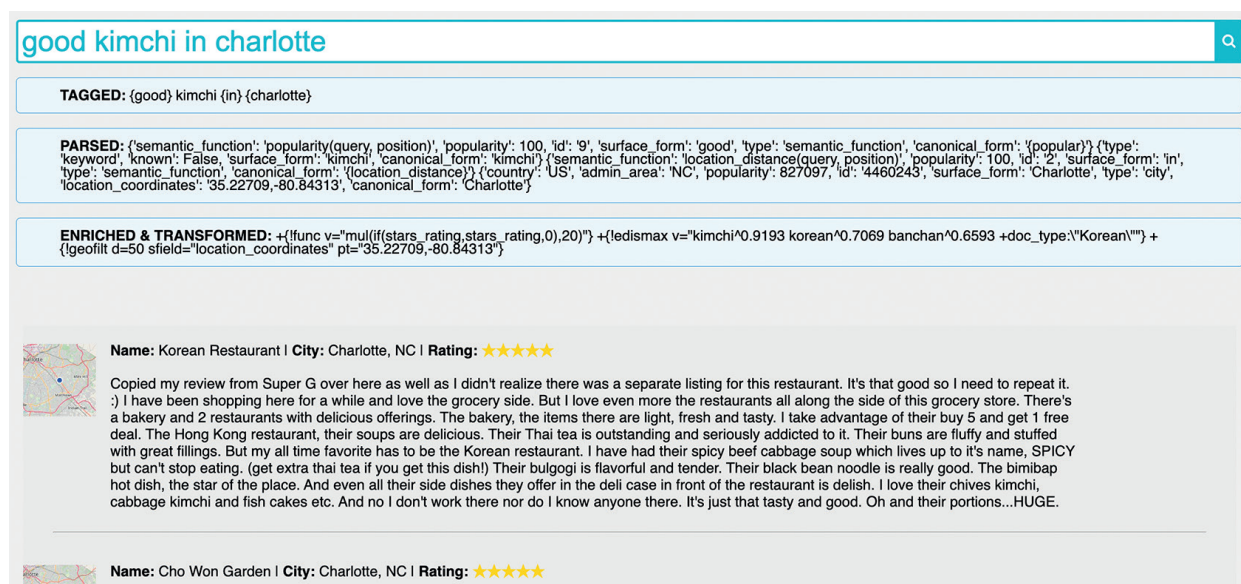


Рис. 7.8. Результаты поиска по запросу good kimchi in charlotte, интерпретируемые как семантически идентичные результатам для top kimchi near charlotte, сквозной пример от начала до конца

Поздравляем, теперь вы реализовали сквозной семантический поисковый конвейер, который может семантически анализировать, обогащать, преобразовывать и выполнять поиск. В этой главе не были представлены какие-либо новые модные алгоритмы машинного обучения, но вместо этого была предоставлена конкретная реализация того, как многие модели, алгоритмы и другие методы, которые вы учили в книге, могут быть интегрированы в сквозную систему.

В оставшейся части книги мы продолжим изучать более продвинутые подходы, которые могут быть включены в эту структуру для повышения релевантности ранжирования и улучшения понимания намерения запроса.

Резюме

- Интерпретация запроса требует надлежащего смешивания конвейеров запроса с изученными моделями, обеспечивая при этом адекватную резервную модель для сопоставления неизвестных ключевых слов.
- Сопоставление только по ключевым словам иногда может работать, но приводит к плохому соответствию, когда намерение, выраженное связанными словами, не понято (top, near и т. д.). Одним из способов решения этой проблемы является реализация домен-специфичных семантических функций для преодоления этих ограничений.
- Семантический парсер запросов, который знает известные термины и фразы, изученные в вашем домене, позволяет вам перейти от поиска на основе ключевых слов к семантическому поиску по сущностям и их связям.

- Извлечение известных сущностей обеспечивает бесшовную интеграцию моделей в конвейер интерпретации вашего запроса, используя сопоставления между поверхностными формами ключевых слов и каноническими представлениями сущностей, сгенерированных из ваших изученных моделей (бустинг сигналов, альтернативные варианты написания, связанные термины и другие данные графа знаний).
- Семантический поиск включает в себя парсинг известных сущностей, обогащение изученными моделями, преобразование запроса для оптимизации соответствия и релевантности для целевой поисковой системы, а затем поиск и возврат результатов конечному пользователю. Мы продолжим изучать более продвинутые методы для включения в эти фазы в следующих главах.

Часть 3

Отраженный интеллект

Поисковый рейтинг не должен быть статической функцией. Каждый запрос и взаимодействие пользователя с результатами поиска – это сигнал, который можно использовать для повышения релевантности будущих результатов. В предыдущей части мы извлекли домен-специфичные знания из контента и сигналов и использовали эти знания для интерпретации намерения запроса.

В этой части мы более подробно рассмотрим тему отраженного интеллекта, процесс обучения на основе взаимодействия пользователя с результатами поиска для улучшения моделей ранжирования релевантности. Мы расширим наше освещение краудсорсинговых алгоритмов релевантности из главы 4, посвятив главы трем ключевым типам отраженного интеллекта: популяризированной релевантности (бустинг сигналов), персонализированной релевантности (совместная фильтрация) и обобщенной релевантности (обучение ранжированию).

Глава 8 рассматривает различные подходы к бустингу сигналов для лучшего ранжирования ваших самых популярных запросов. Затем глава 9 использует сигналы для создания более персонализированного ранжирования релевантности, удовлетворяя особые интересы ваших пользователей на основе их поведения. Глава 10 начинается путешествие из трех глав в мир машинно-обученного ранжирования, также известного как «обучение ранжированию». В главе 10 вы узнаете, как построить модель классификатора ранжирования из суждений о релевантности и как развернуть эту модель в качестве обобщенного алго-

ритма ранжирования релевантности. Глава 11 познакомит вас с моделями кликов, использующими поиск пользователя и сигналы кликов для изучения неявных суждений о релевантности, – и использует эти модели для автоматизации процесса обучения ранжированию. Наконец, глава 12 познакомит вас с подходами активного обучения и А/В-тестирования для преодоления предвзятости в этих неявно обученных моделях путем разумного изучения ранее невиданных результатов и сбора отзывов пользователей в реальном времени.

8

Модели бустинга сигналов

В этой главе рассматривается:

- агрегация сигналов пользователей для создания модели ранжирования на основе популярности;
- нормирование сигналов для шумного ввода запросов;
- борьба со спамом сигналов в краудсорсинговых сигналах;
- применение временного старения для приоритизации¹ последних сигналов;
- объединение нескольких типов сигналов в одну модель;
- выбор между бустингом во время запроса или бустингом во время индексирования.

В главе 4 мы рассмотрели три различные категории отраженного интеллекта: бустинг сигналов (популяризованная релевантность), совместную фильтрацию (персонализированная релевантность) и обучение ранжированию (обобщенная релевантность). В этой главе мы более подробно рассмотрим первую из них, реализуя бустинг сигнала.

¹ Приоритизация – процесс определения и упорядочивания задач и дел в порядке их важности и срочности. Также это действие по назначению приоритетов, присвоение приоритетов кому-либо или чему-либо. – *Прим. ред.*

лов для повышения релевантности рейтинга ваших самых популярных запросов и документов.

В большинстве поисковых систем относительно небольшое количество запросов, как правило, составляет большую часть общего объема запросов. Эти популярные запросы, называемые *головными запросами* (head queries), также, как правило, приводят к большому количеству сигналов (таких как клики и покупки в случае использования электронной коммерции), которые позволяют делать более сильные выводы о популярности топовых результатов поиска.

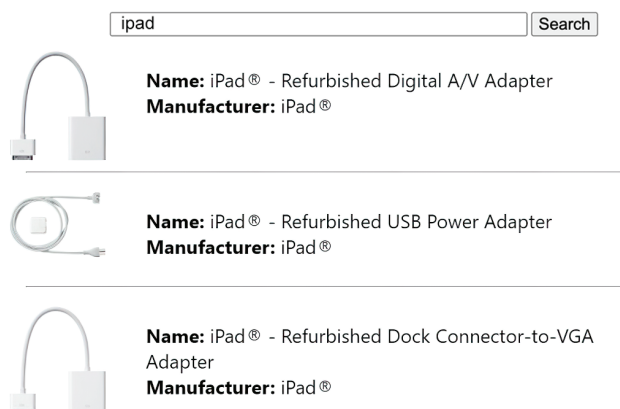
Модели бустинга сигналов напрямую используют эти более сильные выводы и являются ключом к обеспечению того, чтобы ваши самые важные и наиболее заметные запросы были наилучшим образом настроены на возврат наиболее релевантных документов.

8.1. Базовый бустинг сигналов

В разделе 4.2.2 мы построили нашу первую модель бустинга сигналов на основе набора данных RetroTech, что позволило значительно повысить релевантность для наиболее часто искомых и кликабельных результатов поиска. В этом разделе мы кратко рассмотрим процесс создания простой модели бустинга сигналов, которую будем развивать в следующих разделах для решения некоторых более сложных задач.

Вы помните из раздела 4.2.2, что модели бустинга сигналов агрегируют полезные сигналы поведения пользователя в документах (такие как сигналы кликов), которые возникают в результате определенного запроса. Мы использовали поиск по запросу `ipad` и усилили каждый документ на основе того, сколько раз он был ранее кликнут в результатах поиска. Рисунок 8.1 демонстрирует результаты поиска до (нет бустинга сигналов) и после (бустинг сигналов включен) для запроса `ipad`.

До базового бустинга сигналов



После базового бустинга сигналов



Рис. 8.1. До и после применения модели бустинга сигналов. Бустинг сигналов повышает релевантность, перемещая самые популярные предметы в верхнюю часть результатов поиска

Модель бустинга сигналов, которая привела к повышению релевантности на рис. 8.1, является базовой моделью бустинга сигналов. Она рассматривает каждый документ, когда-либо кликнувший для данного запроса, и применяет бустинг, равный общему количеству прошлых кликов по этому документу для этого запроса.

Хотя эта базовая модель бустинга сигналов, описанная в разделе 4.2.2, обеспечивает значительное повышение релевантности, она, к сожалению, подвержена некоторой предвзятости к сигналам и даже манипуляциям. В разделе 8.2 мы обсудим некоторые методы удаления шума в сигналах, чтобы максимизировать качество ваших моделей бустинга сигналов и снизить вероятность нежелательных искажений смысла.

8.2. Нормирование сигналов

Важно нормировать входящие запросы пользователей перед агрегацией, чтобы вариации обрабатывались как один и тот же запрос. Учитывая, что конечные пользователи могут вводить любой произвольный текст в качестве запроса, агрегированные сигналы по своей сути являются шумными. Базовая модель бустинга сигналов из главы 4 (и повторенная в разделе 8.1) не выполняет нормирование. Она генерирует агрегированные бусты для каждой пары запроса и документа, но, поскольку входящие запросы не были нормированы в общую форму, вариации запроса будут обрабатываться как совершенно отдельные запросы. Следующий листинг создает список лучших запросов, которые повышают самую популярную модель iPad в результатах поиска.

Листинг 8.1. Агрегация сигналов
и извлечение релевантных запросов

<pre>def create_boosting_collection(collection_name): basic_signals_aggregation_query = """ SELECT q.target AS query, c.target AS doc, COUNT(c.target) AS boost FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id WHERE c.type = 'click' AND q.type = 'query' GROUP BY q.target, doc ORDER BY boost DESC """ collection = engine.get_collection(collection_name) return aggregate_signals(collection, "basic_signals_boosts", basic_signals_aggregation_query)</pre>	Опре- деляет за- прос агре- гации сигна- лов. Запускает агре- гацию из кол- лекции сигналов в коллекцию basic_signals_ boosts.
--	---

```
def search_for_boosts(query, collection, query_field="query"):
    boosts_request = {"query": query,
                      "query_fields": [query_field],
                      "return_fields": ["query", "doc", "boost"],
                      "limit": 20, 3((C01-16))
                      "order_by": [("boost", "desc")]}
    response = collection.search(**boosts_request)
    return response["docs"]
```

Загружает
бустинги
сигналов
для указан-
ного запро-
са и кол-
лекции.

Самая популярная модель iPad.

```
signals_boosting_collection = create_boosting_collection("signals")
query = "885909457588"
signals_docs = search_for_boosts(query, signals_boosting_collection, "doc")
show_raw_boosted_queries(signals_docs)
```

Возвращает список лучших бустов сигналов для указанного документа.

Вывод:

Raw Boosted Queries

```
"iPad" : 1050
"ipad" : 966
"Ipad" : 829
"iPad 2" : 509
"ipad 2" : 347
"Ipad2" : 261
"ipad2" : 238
"Ipad 2" : 213
"I pad" : 203
"i pad" : 133
"IPad" : 77
"Apple" : 76
"I pad 2" : 60
"apple ipad" : 55
"Apple iPad" : 53
"ipads" : 43
"tablets" : 42
"apple" : 41
"iPads" : 38
"i pad 2" : 38
```

Этот список объединяет все сигналы по их запросам и сохраняет каждый запрос вместе с числом вхождений в новую коллекцию. Из вывода видно, что в базовой модели бустинга сигналов существует множество вариаций одних и тех же запросов. Главной причиной вариаций, по-видимому, является чувствительность к регистру, поскольку мы видим iPad, ipad, Ipad и IPad как распространенные варианты. Интервалы, по-видимому, являются еще одной проблемой, с ipad 2, I pad2 и Ipad2. Мы даже видим единственное число ipad и множественное ipads.

Поля поиска ключевых слов обычно нормируют запросы, чтобы они были нечувствительны к регистру, используют стемминг¹ для игнорирования множественных версий терминов и разделяют по изменениям регистра и переходам букв в числа между словами. Также полезно нормировать сигналы, поскольку сохранение отдельных терминов запроса и усилений (boosts) для вариантов, которые неразличимы поисковой системой, может быть контрпродуктивным. Невозможность нормировать термины рассеивает ценность ваших сигналов, поскольку сигналы делятся на вариации тех же ключевых слов с более низкими усилениями, а не объединяются в более значимые запросы с более сильными бустами.

Вы должны выяснить, насколько сложным должно быть нормирование вашего запроса до агрегации сигналов, но даже простое приведение входящих запросов к нижнему регистру, чтобы сделать агрегацию сигналов нечувствительной к регистру, может иметь большое значение. Следующий листинг демонстрирует ту же базовую агрегацию сигналов, что и раньше, но на этот раз с запросами, приведенными к нижнему регистру.

Листинг 8.2. Агрегация сигналов без учета регистра

```
normalized_signals_aggregation_query = """
SELECT LOWER(q.target) AS query,
c.target AS doc, COUNT(c.target) AS boost
FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
WHERE c.type = 'click' AND q.type = 'query'
GROUP BY LOWER(q.target), doc
ORDER BY boost DESC
"""

normalized_collection = \
    aggregate_signals(signals_collection, "normalized_signals_boosts",
                     normalized_signals_aggregation_query)

query = "885909457588"
signals_documents = search_for_boosts(query, normalized_collection, "doc")
show_raw_boosted_queries(signals_documents)
```

Нормирование регистра путем перевода каждого запроса в нижний регистр.

Группировка по нормированному запросу увеличивает количество сигналов для этих запросов, увеличивая бустинг сигналов.

Самая популярная модель iPad.

Вывод:

```
Raw Boosted Queries
"ipad" : 2939
"ipad 2" : 1104
```

¹ Стемминг в программировании – это процесс сокращения слова до его корневой или базовой формы. Это делается путем удаления суффиксов, префиксов и других инфиксов. Стемминг используется в задачах обработки естественного языка для стандартизации текста и улучшения производительности алгоритмов. Также он может помочь группировать слова с одинаковым значением, даже если они имеют разные формы. – *Прим. ред.*

```
"ipad2" : 540
"i pad" : 341
"apple ipad" : 152
"ipads" : 123
"apple" : 118
"i pad 2" : 99
"tablets" : 67
"tablet" : 61
"ipad 1" : 52
"apple ipad 2" : 27
"hp touchpad" : 26
"ipaq" : 20
"i pad2" : 19
"wi" : 19
"apple computers" : 18
"apple i pad" : 15
"ipad 2 16gb" : 15
"samsung galaxy" : 14
```

Список необработанных усиленных запросов уже выглядит намного чище! Не только стало меньше избыточности, но и обратите внимание, что сила бустинга сигналов увеличилась, поскольку больше сигналов приписывается канонической форме запроса (версии в нижнем регистре).

Строчные буквы в запросах и, возможно, удаление пробелов или посторонних символов часто являются достаточным нормированием для запросов перед агрегированием сигналов. Однако важный вывод из этого раздела заключается в том, что модель бустинга сигналов становится тем сильнее, чем больше вы можете гарантировать, что идентичные запросы рассматриваются как одинаковые при агрегировании.

Различия в запросах не единственный вид шума, о котором нам нужно беспокоиться в наших данных. В следующем разделе мы поговорим о том, как можно преодолеть значительные потенциальные проблемы, вызванные спамом в наших сигналах кликов, сгенерированных пользователями.

8.3. Борьба со спамом сигналов

Всякий раз, когда мы используем краудсорсинговые данные, такие как сигналы кликов, для влияния на поведение поисковой системы, нам нужно спросить себя: «Как можно манипулировать входными данными, чтобы создать нежелательный результат?» В этом разделе мы покажем, как можно спамить поисковую систему сигналами кликов для манипулирования результатами поиска и как это можно остановить.

8.3.1. Использование спама сигналов для манипулирования результатами поиска

Представим, что у нас есть пользователь, который по какой-то причине действительно ненавидит «Звездные войны» и считает, что последние фильмы – полный мусор. Фактически такие пользователи

чувствуют это настолько сильно, что хотят гарантировать, что любой поиск по запросу `star wars` возвращает полный треш для покупки в качестве верхнего результата поиска. Этот пользователь кое-что знает о поисковых системах и заметил, что ваши убийственные алгоритмы релевантности, похоже, используют пользовательские сигналы и бустинг сигналов. На рис. 8.2 показан ответ по умолчанию для запроса `star wars`, при этом бустинг сигналов выводит самые популярные продукты наверх результатов поиска.

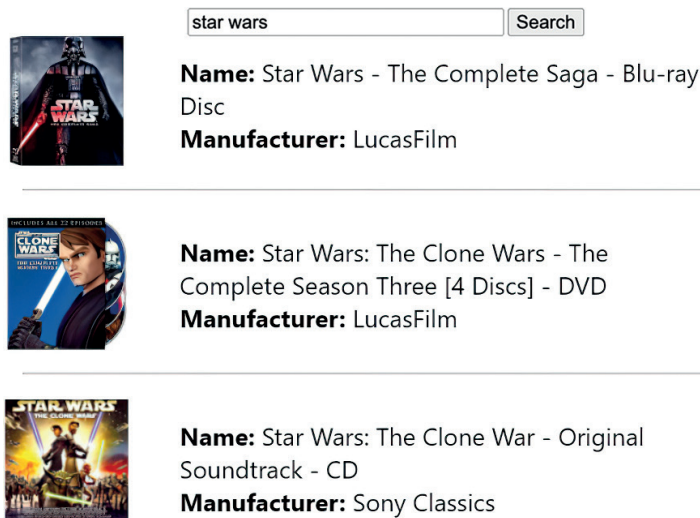


Рис. 8.2. Самые популярные результаты поиска по запросу `star wars` с включенным бустингом сигналов. Это ожидаемые результаты, когда нет вредоносного спама сигнала

Пользователь решает, что, поскольку рейтинг вашей поисковой системы основан на популярных предметах, он будет спамить поисковую систему кучей поисковых запросов `star wars`. Он будет сопровождать каждый поиск поддельными кликами по найденной мусорной корзине в стиле «Звездные войны», пытаясь сделать так, чтобы мусорная корзина отображалась в верхней части результатов поиска.

Чтобы смоделировать этот сценарий, в следующем листинге мы запустим простой скрипт, чтобы сгенерировать 5000 запросов `star wars` и 5000 соответствующих кликов по мусорной корзине после выполнения этого запроса.

Листинг 8.3. Генерация спам-запросов и сигналов кликов

```
signals_collection = engine.get_collection("signals")
spam_user = "u8675309"
spam_query = "star wars"
spam_signal_boost_doc_upc = "45626176"
```

Документ для корзины, которую спамер хочет переместить в начало результатов поиска.

```
signal_docs = []
for num in range(5000):
    query_id = f"u8675309_0_{num}"
    query_signal = {
```

Генерирует и отправляет 5000 запросов и сигналов кликов в поисковую систему.

```

    "query_id": query_id,
    "user": spam_user,
    "type": "query",
    "target": spam_query,
    "signal_time": datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"),
    "id": f"spam_signal_query_{num}"
click_signal = {
    "query_id": query_id,
    "user": spam_user,
    "type": "click",
    "target": spam_signal_boost_doc_upc,
    "signal_time": datetime.now().strftime("%Y-%m-%dT%H:%M:%SZ"),
    "id": f"spam_signal_click_{num}"
signal_docs.extend([click_signal, query_signal])

signals_collection.add_documents(signal_docs)

spam_signals_collection = \
    aggregate_signals(signals_collection, "signals_boosts_with_spam",
        normalized_signals_aggregation_query)

```

Генерирует и отправляет 5000 запросов и сигналов кликов в поисковую систему.

Запускает агрегацию сигналов для создания модели бустинга сигналов, включающей сигналы спама.

Листинг 8.3 отправляет тысячи спам-запросов и сигналов кликов в нашу поисковую систему, моделируя тот же результат, который мы увидели бы, если бы пользователь искал и кликал по определенному результату поиска тысячи раз. Затем листинг повторно запускает базовую агрегацию сигналов.

Чтобы увидеть влияние спам-кликов злоумышленника на наши результаты поиска, следующий листинг запускает поиск по запросу `star wars`, теперь включающему манипулированную модель бустинга сигналов.

Листинг 8.4. Результаты поиска, зашумленного сигналами спам-пользователя

```

def boosted_product_search_request(query, collection, boost_field=None):
    signals_documents = search_for_boosts(query, collection)
    signals_boosts = create_boosts_query(signals_documents)
    boosted_request = product_search_request(query)
    if boost_field:
        signals_boosts = (boost_field, signals_boosts)
        boosted_request["query_boosts"] = signals_boosts
    return boosted_request

query = '"star wars"'
boosted_request = boosted_product_search_request(query,
    spam_signals_collection, "upc")

```

Загружает бустинг сигналов из модели бустинга сигналов, включающей спам-сигналы.

Усиливает запрос «Звездные войны» с помощью модели бустинга сигналов.


```
response = products_collection.search(**boosted_request)
display_product_search(query, response["docs"])
```

На рис. 8.3 показаны новые измененные результаты поиска, сгенерированные из листинга 8.4, с мусорной корзиной «Звездных войн» на первом месте.

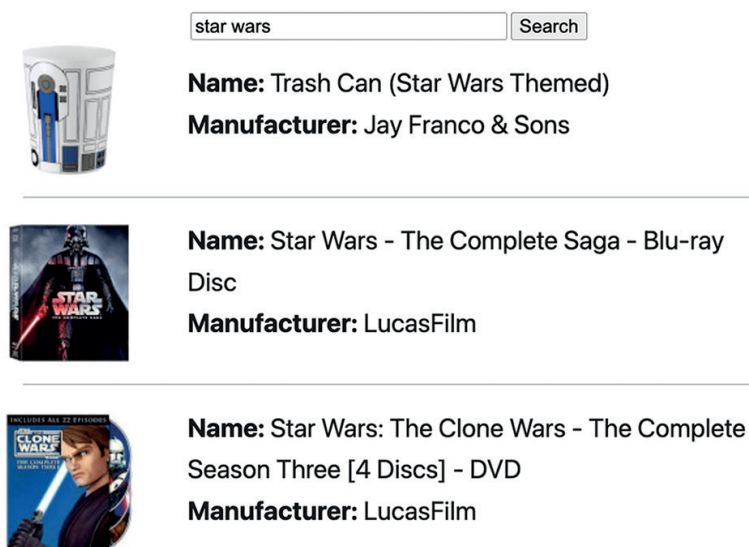


Рис. 8.3. Результаты поиска, измененные пользователем, заспамившим поисковую систему поддельными сигналами, чтобы повлиять на топовый результат. Пользователь смог изменить его, просто кликнув по нему много раз

Спамер добился успеха, и эти измененные результаты поиска теперь будут видны каждому последующему посетителю, который ищет `star wars` на веб-сайте RetroTech! Похоже, нам нужно сделать нашу модель бустинга сигналов более надежной для борьбы с этим видом спама сигналов.

8.3.2. Борьба со спамом сигналов с помощью фильтрации на основе пользователей

Если вы собираетесь использовать краудсорсинговые данные, такие как сигналы пользователей, для влияния на рейтинг поисковой системы, важно предпринять шаги, чтобы свести к минимуму возможность пользователей манипулировать вашим алгоритмом ранжирования на основе сигналов.

Чтобы справиться с проблемой мусорной корзины «Звездных войн», которую мы только что продемонстрировали, самый простой метод – гарантировать, что повторяющиеся клики одного и того же пользователя получают только один «голос» в агрегации бустинга сигналов. Таким образом, независимо от того, нажимает ли злонамеренный пользователь один раз или миллион раз, его клики считаются только одним сигналом и, следовательно, не оказывают существенного влияния на модель бустинга сигналов. Следующий листинг перерабатывает запрос агрегации сигналов, чтобы подсчитывать только уникальные сигналы кликов от каждого пользователя.

Листинг 8.5. Дедупликация сигналов шума пользователя

```

anti_spam_aggregation_query = """
SELECT query, doc, COUNT(doc) AS boost FROM (
    SELECT c.user unique_user, LOWER(q.target) AS query, c.target AS doc,
    MAX(c.signal_time) AS boost
    FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
    WHERE c.type = 'click' AND q.type = 'query'
    GROUP BY unique_user, LOWER(q.target), doc)
GROUP BY query, doc
ORDER BY boost DESC"""

```

Группируйте по пользователю, чтобы ограничить каждого пользователя одним «голосом» на пару запрос/документ в модели бустинга сигналов.

Дата сигнала – это самый последний сигнал от пользователя, если есть дубликаты.

```

anti_spam_collection = \
    aggregate_signals(signals_collection, "signals_boosts_anti_spam",
        anti_spam_aggregation_query)

```

Если мы повторно запустим запрос `star wars` из листинга 8.3 с этой новой моделью `signals_boosts_anti_spam`, мы увидим, что наши обычные результаты поиска вернулись и выглядят так же, как на рис. 8.2. Это связано с тем, что все дополнительные спам-сигналы от нашего злонамеренного пользователя были сведены к одному плохому сигналу, как показано в табл. 8.1.

Таблица 8.1. 5000 спам-сигналов были дедуплицированы до 1 сигнала в антиспам-модели бустинга сигналов

model	query	doc	boost
До спам-сигналов (normalized_signals_boosts)	star wars	400032015667	0 (пока нет сигналов)
После спам-сигналов (normalized_signals_boosts)	star wars	400032015667	5000
После очистки спам-сигналов (signals_boosts_anti_spam)	star wars	400032015667	1

Вы можете видеть, что агрегированное количество сигналов в модели `signals_boosts_anti_spam` имеет общее значение, гораздо более близкое к модели `normalized_signals_boosts`, которую мы построили до генерации спам-сигналов. Поскольку каждый пользователь ограничен одним сигналом на пару запрос/документ в модели `signals_boosts_anti_spam`, возможность пользователей манипулировать моделью бустинга сигналов теперь существенно снижена.

Конечно, вы можете определить любые учетные записи пользователей, которые, по-видимому, спамят вашу поисковую систему, и полностью удалить их сигналы из агрегации бустинга сигналов, но сокращение охвата сигналов посредством дедупликации проще и часто достигает той же конечной цели восстановления хорошего кра-

удсорсингового рейтинга релевантности. В листинге 8.5 мы использовали ID пользователей в качестве ключевого ID для дедупликации спам-сигналов, но здесь подойдет любой ID: ID пользователя, ID сеанса, ID браузера, IP-адрес или даже какой-то отпечаток браузера. Если вы найдете какой-то ресурс для уникальной идентификации пользователей или для идентификации трафика низкого качества (например, ботов и веб-скрейперов), вы можете использовать эту информацию для дедупликации сигналов. Если ни один из этих методов не работает и в ваших сигналах кликов слишком много шума, вы также можете выбрать просмотр только сигналов кликов от известных (аутентифицированных) пользователей, которые, как вы уверенно можете предположить, являются легитимными пользователями вашей системы.

Один из последних способов смягчить проблему спама сигналов – найти способ отделить важные типы сигналов от шумных, которыми можно легко манипулировать. Например, генерировать сигналы от запуска запросов и клика на результаты поиска легко. Сигналами от покупки продукта гораздо сложнее манипулировать, поскольку они требуют от пользователей входа в систему или ввода платежной информации, прежде чем покупка будет логирована. Вероятность того, что кто-то злонамеренно купит 5000 мусорных баков «Звездные войны», довольно мала, поскольку для этого существует множество финансовых и логистических барьеров. Не только ценно взвешивать покупки как более сильные сигналы, чем клики, с точки зрения борьбы со спамом, но и ценно с точки зрения релевантности, потому что покупки являются более четкими индикаторами намерения. В следующем разделе мы рассмотрим, как можно объединить различные типы сигналов в модель бустинга сигналов, которая учитывает относительную важность каждого типа сигнала.

8.4. Объединение нескольких типов сигналов

До сих пор мы работали только с двумя типами сигналов – запросами и кликами. Для некоторых поисковых систем (например, поисковых систем в интернете) сигналы кликов могут быть единственным хорошим источником краудсорсинговых данных, доступных для построения модели бустинга сигналов. Однако существует много различных типов сигналов, которые могут предоставить дополнительные и часто гораздо лучшие входные данные для построения модели бустинга сигналов.

В нашем наборе данных RetroTech есть несколько видов сигналов, которые являются общими для вариантов использования электронной коммерции:

- запрос,
- клик,
- добавить в корзину,
- купить.

Хотя клики в ответ на запросы полезны, они не обязательно подразумевают сильный интерес к продукту, так как кто-то может просто

просматривать, чтобы увидеть, что доступно. Если кто-то добавляет продукт в свою корзину, это обычно представляет собой гораздо более сильный сигнал интереса, чем клик. Покупка является еще более сильным сигналом того, что пользователь заинтересован в продукте, так как пользователь готов заплатить деньги, чтобы получить продукт, который он искал.

В то время как некоторые веб-сайты электронной коммерции могут получать достаточно трафика, чтобы полностью игнорировать сигналы кликов и сосредоточиться только на сигналах добавления в корзину и покупки, часто бывает полезнее включать все типы сигналов при расчете бустинга сигналов. К счастью, объединение нескольких типов сигналов так же просто, как назначение относительных весов в качестве множителей для каждого типа сигнала при выполнении агрегации сигналов:

```
signals_boost = (1 * sum(click_signals)) +
                 (10 * sum(add_to_cart_signals)) +
                 (25 * sum(purchase_signals))
```

Считая каждый клик как 1 сигнал, каждое добавление в корзину как 10 сигналов и каждую покупку как 25 сигналов, покупка несет в 25 раз больший вес, а добавление в корзину несет в 10 раз больший вес, чем клик в модели бустинга сигналов. Это помогает уменьшить шум от менее надежных сигналов и усиливает более надежные сигналы, при этом по-прежнему используя большой объем менее надежных сигналов в случаях, когда лучшие сигналы менее распространены (например, новые или малоизвестные продукты).

В следующем листинге показано агрегирование сигналов с различными весами, предназначенное для объединения различных типов сигналов.

Листинг 8.6. Объединение нескольких типов сигналов с разными весами

```
mixed_signal_types_aggregation_query = """
SELECT query, doc, ((1 * click_boost)
    + (10 * add_to_cart_boost) +
    (25 * purchase_boost)) AS boost FROM (
    SELECT query, doc,
        SUM(click) AS click_boost,
        SUM(add_to_cart) AS add_to_cart_boost,
        SUM(purchase) AS purchase_boost FROM (
        SELECT lower(q.target) AS query, cap.target AS doc,
            IF(cap.type = 'click', 1, 0) AS click,
            IF(cap.type = 'add-to-cart', 1, 0) AS add_to_cart,
            IF(cap.type = 'purchase', 1, 0) AS purchase
        FROM signals cap LEFT JOIN signals q on cap.query_id = q.query_id
        WHERE (cap.type != 'query' AND q.type = 'query')
    ) raw_signals
    GROUP BY query, doc) AS per_type_boosts"""
```

Несколько сигналов объединяются с различными относительными весами для расчета общего значения бустинга.

Каждый тип сигнала суммируется независимо перед объединением.

```

type_weighted_collection = \
    aggregate_signals(signals_collection, "signals_boosts_weighted_
types",
                    mixed_signal_types_aggregation_query)

```

Из SQL-запроса видно, что общий прирост для каждой пары запрос/документ рассчитывается путем подсчета всех кликов с весом 1, подсчета всех сигналов добавления в корзину и умножения их на вес 10, а также подсчета всех сигналов покупки и умножения их на вес 25.

Эти предлагаемые веса 10х для сигналов добавления в корзину и 25х для сигналов покупки должны хорошо работать во многих сценариях электронной коммерции, но эти относительные веса также полностью настраиваются для каждого домена. Ваш веб-сайт может быть настроен таким образом, что почти каждый, кто добавляет продукт в свою корзину, покупает продукт (например, приложение для доставки продуктов из продовольственного магазина, где единственная цель использования веб-сайта – заполнить корзину и совершить покупку). В этих случаях вы можете обнаружить, что добавление продукта в корзину не добавляет никакой дополнительной ценности, но удаление продукта из корзины может потенциально повлечь за собой штраф, указывая на то, что продукт плохо соответствует запросу.

В этом случае вы можете захотеть ввести идею *бустинга негативных сигналов*. Так же как мы обсуждали клики, добавления в корзину и покупки как сигналы намерения пользователя, ваш пользовательский опыт также может иметь множество способов измерения неудовлетворенности пользователя результатами поиска. Например, у вас может быть кнопка «не нравится» или кнопка «удалить из корзины», или вы можете отслеживать возвраты продукта после покупки. Вы даже можете захотеть подсчитать документы в результатах поиска, которые были пропущены, и записать сигнал «пропустить» для этих документов, чтобы указать, что пользователь их видел, но не проявил интереса. Мы рассмотрим тему управления кликнутыми и пропущенными документами в главе 11, когда будем обсуждать моделирование кликов.

К счастью, обрабатывать отрицательные отзывы так же просто, как и обрабатывать положительные сигналы: вместо того чтобы просто назначать все более положительные веса сигналам, вы также можете назначать все более отрицательные веса отрицательным сигналам. Вот пример:

```

positive_signals = (1 * sum(click_signals)) +
                  (25 * sum(purchase_signals)) +
                  (10 * sum(add_to_cart_signals)) +
                  (0.025 * sum(seen_doc_signals))
negative_signals = (-0.025 * sum(skipped_doc_signals)) +
                  (-20 * sum(remove_from_cart_signals)) +
                  (-100 * sum(returned_item_signals)) +
                  (-50 * sum(negative_post_about_item_in_review_
signals))
type_based_signal_weight = positive_signals + negative_signals

```


Эта простая линейная функция обеспечивает высоконастраиваемую модель ранжирования на основе сигналов, принимая несколько входных параметров и возвращая оценку ранжирования на основе относительных весов этих параметров. Вы можете объединить столько полезных сигналов, сколько захотите, в эту агрегацию взвешенных сигналов, чтобы повысить надежность модели. Конечно, настройка весов каждого из типов сигналов для достижения оптимального баланса может потребовать некоторых усилий. Вы можете сделать это вручную или использовать для этого метод машинного обучения, называемый обучением ранжированию. Мы рассмотрим обучение ранжированию в главах 10 и 11.

Важно не только взвешивать различные виды сигналов относительно друг друга, но иногда также может быть необходимо взвешивать *один и тот же* вид сигналов по-разному относительно друг друга. В следующем разделе мы обсудим один ключевой пример этого: присвоение более высокого значения более поздним по времени взаимодействиям.

8.5. Временные спады и короткоживущие сигналы

Сигналы не всегда сохраняют свою полезность бесконечно. В последнем разделе мы показали, как можно настроить модели бустинга сигналов, чтобы взвешивать различные виды сигналов как более важные, чем другие. В этом разделе мы рассмотрим другую задачу – учет «временной ценности» сигналов по мере их старения и снижения полезности.

Представьте себе три различных варианта использования поисковой системы:

- поисковая система электронной коммерции со стабильными продуктами,
- поисковая система вакансий,
- новостной сайт.

Для поисковой системы электронной коммерции, такой как Ret-goTech, документы (продукты) часто остаются на протяжении многих лет, и лучшими продуктами часто являются те, которые имеют долгую историю интереса.

В поисковой системе вакансий документы (вакансии) могут оставаться на месте только несколько недель или месяцев, пока вакансия не будет заполнена, а затем они исчезают навсегда. Однако, хотя документы присутствуют, новые клики или заявления о приеме на работу не обязательно так же важны, как сигналы, чем старые взаимодействия.

В поисковой системе новостей, хотя новостные статьи остаются навсегда, новые статьи, как правило, намного важнее старых, и новые сигналы также важнее старых сигналов, поскольку интересы людей меняются ежедневно, если не ежечасно.

Давайте углубимся в эти варианты использования и покажем, как лучше всего обрабатывать временную ценность сигналов при выполнении их бустинга.

8.5.1. Обработка нечувствительных ко времени сигналов

В нашем варианте использования RetroTech наши документы намеренно старые, они существуют уже десять лет или больше, и интерес к ним, вероятно, только увеличивается по мере того, как продукты становятся старше и более «ретро». В результате у нас не часто бывают резкие скачки популярности предметов, а новые сигналы не обязательно имеют значительно большую важность, чем старые сигналы. Этот тип варианта использования немного нетипичен, но многие варианты использования поиска действительно имеют дело с такими «статичными» наборами документов. Лучшим решением в этом случае является стратегия, которую мы уже использовали в этой главе: обработать все сигналы в течение разумного периода месяцев или лет и придать им одинаковый вес. Когда все периоды времени имеют одинаковый вес, модель бустинга сигналов, вероятно, не нужно будет перестраивать очень часто, поскольку модель медленно меняется с течением времени. Частая обработка сигналов – это ненужные вычислительные издержки.

Однако в случае поиска работы сценарий совсем другой. Ради аргумента предположим, что в среднем требуется 30 дней, чтобы закрыть вакансию. Это означает, что документ, представляющий эту вакансию, будет присутствовать в поисковой системе только в течение 30 дней, и любые сигналы, собранные для этого документа, полезны только для бустинга сигналов в течение этого 30-дневного окна. Когда вакансия публикуется, она, как правило, будет очень популярной в течение первых нескольких дней, поскольку она новая и, вероятно, привлечет много существующих соискателей, но все взаимодействия с этой вакансией в любой момент в течение 30 дней так же полезны. В этом случае все сигналы кликов должны получить одинаковый вес, и все сигналы заявлений о приеме на работу также должны получить одинаковый вес (с весом выше, чем сигналы кликов). Однако, учитывая очень короткий срок службы документов, важно, чтобы все сигналы обрабатывались как можно быстрее, чтобы максимально использовать их ценность.

Варианты использования с недолговечными документами, как в случае поиска работы, обычно не являются лучшими кандидатами для бустинга сигналов, поскольку документы могут быть удалены к тому времени, когда модель бустинга сигналов станет хоть сколько-нибудь эффективной. В результате часто может быть более разумным рассмотреть персонализированные модели (например, совместную фильтрацию, описанную в главе 9) и обобщаемые модели релевантности (например, обучение ранжированию, описанное в главах 10 и 11) для этих вариантов использования.

Как в случае RetroTech, так и в случае поиска работы сигналы были одинаково полезны в течение всего срока существования документа. В случае поиска новостей, который мы рассмотрим далее, ценность сигналов со временем снижается.

8.5.2. Обработка сигналов, чувствительных ко времени

В случае использования поисковой системы новостей самые последние опубликованные новости обычно получают наибольшее взаимодействие, поэтому более свежие сигналы значительно ценнее, чем старые сигналы. Некоторые новости могут быть очень популярны и актуальны в течение нескольких дней или дольше, но, как правило, сигналы за последние десять минут более ценны, чем сигналы за последний час, которые более ценны, чем сигналы за последний день, и т. д. Поиск новостей – это экстремальный случай использования, когда сигналы должны обрабатываться быстро, а более свежие сигналы должны быть взвешены как существенно более важные, чем старые сигналы.

Один из простых способов моделирования этого – использование функции затухания, такой как функция полураспада, которая уменьшает вес, назначенный сигналу, вдвое (50 %) за равноотстоящие промежутки времени. Например, функция спада с периодом полураспада 30 дней присвоит 100%-ный вес сигналу, который происходит «сейчас», 75%-ный вес сигналу 15-дневной давности, 50%-ный вес сигналу 30-дневной давности, 25%-ный вес сигналу 60-дневной давности, 12.5%-ный вес сигналу 90-дневной давности и т. д. Математика для расчета веса сигнала на основе времени с использованием функции затухания:

$$\text{starting_weight} \times 0.5(\text{signal_age} / \text{half_life})$$

При применении этого расчета `starting_weight` обычно будет относительным весом сигнала на основе его типа, например вес 1 для кликов, 10 для сигналов добавления в корзину и 25 для сигналов покупки. Если вы не объединяете несколько типов сигналов, то `starting_weight` будет просто 1.

Параметр `Signal_age` – это возраст сигнала, а `half_life` – это то, сколько времени требуется сигналу, чтобы потерять половину своего значения. На рис. 8.4 показано, как эта функция затухания влияет на веса сигнала с течением времени для различных значений периода полураспада.

Период полураспада в 1 день очень агрессивен и непрактичен в большинстве случаев использования, поскольку маловероятно, что вы сможете собрать достаточно сигналов за день для обеспечения значимого бустинга сигналов, а вероятность того, что ваши сигналы быстро станут неактуальными, мала.

Период полураспада в 30, 60 и 120 дней эффективен для агрессивного снижения старых сигналов, сохраняя при этом остаточную пользу от сниженных сигналов в течение периода от шести до двенадцати месяцев. Если у вас очень долгоживущие документы, вы можете продвинуться еще дальше, используя сигналы в течение многих лет. Следующий листинг демонстрирует обновленный запрос агрегации сигналов, который реализует период полураспада в 30 дней для каждого сигнала.

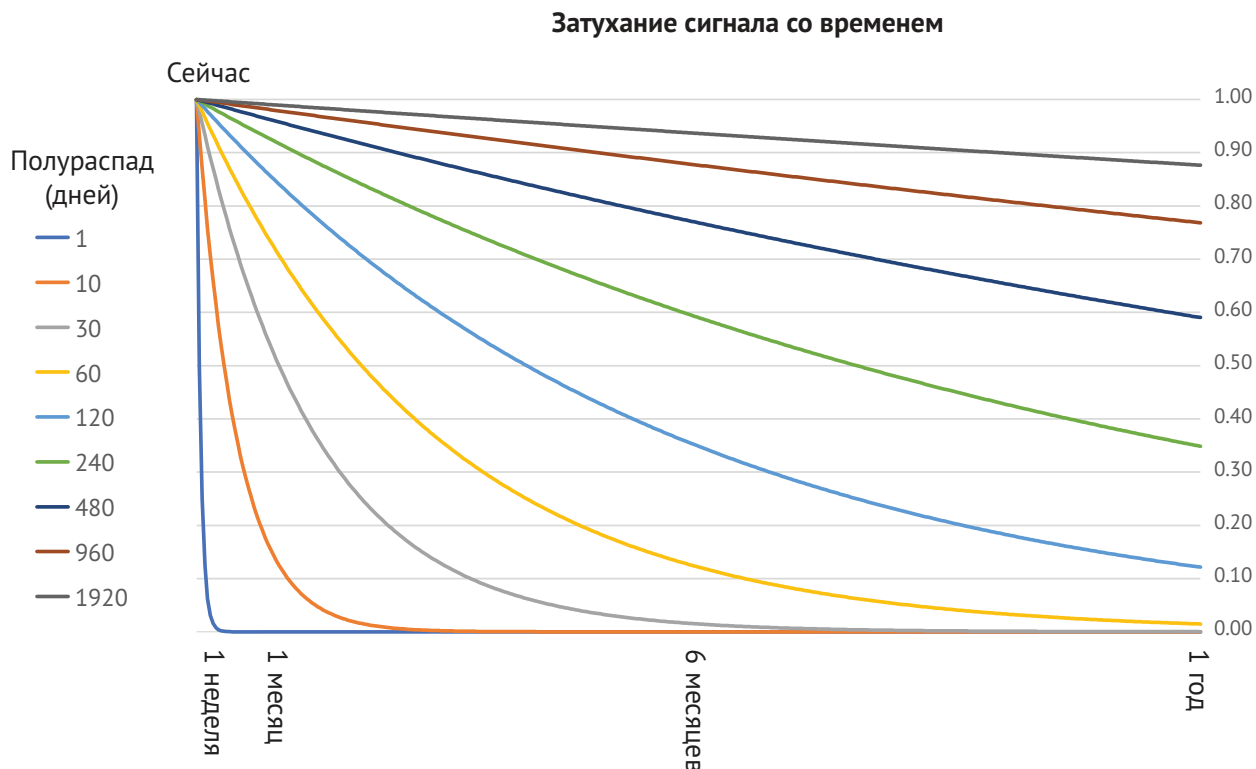


Рис. 8.4. Затухание сигнала с течением времени на основе различных значений периода полураспада. По мере увеличения периода полураспада отдельные сигналы сохраняют свою усиливающую силу дольше

Листинг 8.7. Применение затухания со временем к модели бустинга сигналов

```

half_life_days = 30
target_date = '2024-06-01'
signal_weight = 1

time_decay_aggregation = f"""
SELECT query, doc, sum(time_weighted_boost) AS boost FROM (
  SELECT user, query, doc, {signal_weight} *
    POW(0.5, (DATEDIFF('{target_date}', signal_time) /
      {half_life_days}))
  AS time_weighted_boost FROM (
    SELECT c.user AS user, lower(q.target) AS query, c.target AS doc,
      MAX(c.signal_time) as signal_time
    FROM signals c LEFT JOIN signals q ON c.query_id = q.query_id
    WHERE c.type = 'click' AND q.type = 'query'
    AND c.signal_time <= '{target_date}'
    GROUP BY c.user, q.target, c.target
  ) AS raw_signals
  ) AS time_weighted_signals
GROUP BY query, doc
ORDER BY boost DESC"""

```

Последняя возможная дата сигнала. Это должно быть pow() в живой системе, но ее можно установить на фиксированную дату для замороженного набора данных, такого как RetroTech.

Здесь можно добавить функцию для дифференциации весов для разных типов сигналов.

Расчет периода полураспада.

Включает только сигналы до целевой даты.

Получает самый последний уникальный сигнал для пользователя, запроса и комбинации продуктов.

```
time_weighted_collection = \  
    aggregate_signals(signals_collection, "signals_boosts_time_  
weighted",  
                      time_decay_aggregation)
```

Эта функция распада имеет несколько уникальных настраиваемых параметров:

- она содержит параметр `half_life_days`, который вычисляет средневзвешенное значение с использованием настраиваемого периода полураспада, который мы для начала установили на 30 дней;
- она содержит параметр `signal_weight`, который можно заменить функцией, возвращающей вес по типу сигнала, как показано в последнем разделе («click» = 1, «add-to-cart» = 10, «purchase» = 25 и т. д.);
- она содержит параметр `target_date`, который является датой, когда сигнал получает полное значение 1. Все сигналы до этой даты будут распадаться на основе периода полураспада, а все сигналы после этой даты будут игнорироваться (отфильтровываться).

Обычно `target_date` будет текущей датой, так что вы используете самые последние сигналы и назначаете им самый высокий вес. Однако вы также можете применить его к прошлым периодам, если ваши документы имеют сезонные закономерности, которые повторяются ежемесячно или ежегодно.

Хотя наши документы о продуктах не меняются очень часто, и самые последние сигналы не обязательно являются более ценными, чем старые сигналы, потенциально существуют годовые закономерности, которые мы могли бы найти в обычном наборе данных электронной коммерции. Например, определенные типы продуктов могут быть более популярны в период крупных праздников и дат, таких как День матери, День отца и Черная пятница. Аналогично поисковые запросы по слову «лопата» могут иметь разное значение летом (лопата для копания земли) по сравнению с зимой (лопата для уборки снега с тротуара). Если вы исследуете свои сигналы, может возникнуть множество тенденций, для которых чувствительность ко времени должна влиять на то, как взвешиваются ваши сигналы.

В конечном счете сигналы являются запаздывающим индикатором. Они отражают то, что только что сделали ваши пользователи, но они полезны только в качестве прогнозов будущего поведения, если изученные закономерности будут повторяться, что более или менее вероятно.

Теперь, изучив методы улучшения наших моделей сигналов посредством нормирования запросов, снижения спама и манипуляций релевантностью, объединения нескольких типов сигналов с различными относительными весами и применения временных спадов к сигналам, вы должны быть в состоянии гибко реализовывать модели бустинга сигналов, наиболее подходящие для вашего варианта использования. Однако при развертывании бустинга сигналов в масштабе есть два разных подхода, которые вы можете использовать для оптимизации гибкости и производительности, мы рассмотрим их в следующем разделе.

8.6. Бустинг во время индексирования или во время запроса: балансировка масштаба и гибкости

Все модели бустинга сигналов в этой главе были продемонстрированы с использованием *бустинга во время запроса*, которое делает бустинг сигналов для каждого запроса из отдельной коллекции сайдкаров во время запроса и изменяет запрос, чтобы добавить бусты перед отправкой его в поисковую систему. Также возможно реализовать модели бустинга сигналов с помощью *бустинга во время индексирования*, где усиления делаются непосредственно в документах для запросов, к которым эти усиления применяются. В этом разделе мы покажем вам, как реализовать усиление во время индексирования, и обсудим преимущества и недостатки бустинга во время запроса по сравнению с усилением во время индексирования.

8.6.1. Компромиссы при использовании бустинга во время запроса

Бустинг во время запроса, как мы видели, превращает каждый запрос в двухэтапный процесс. Каждый входящий запрос пользователя ищется в коллекции бустинга сигналов, и любые найденные усиленные документы используются для изменения запроса пользователя. Бустинг во время запроса является наиболее распространенным способом реализации усиления сигналов, но у него есть свои преимущества и недостатки.

Преимущества бустинга во время запроса

Основная архитектурная характеристика бустинга во время запроса заключается в том, что он разделяет основную коллекцию поиска (*products*) и коллекцию бустинга сигналов (**_signals_boosts*). Такое разделение дает несколько преимуществ:

- оно позволяет обновлять сигналы для каждого запроса постепенно, изменяя только один документ, представляющий этот запрос;
- оно позволяет легко включать или выключать бустинг, просто не выполняя поиск или не изменяя запрос пользователя;
- оно позволяет в любое время менять алгоритмы бустинга сигналов.

В конечном счете гибкость изменения бустинга сигналов в любой момент времени на основе текущего контекста является основным преимуществом бустинга сигналов во время запроса. Это позволяет легче включать сигналы в реальном времени и экспериментировать с различными функциями ранжирования.

Недостатки бустинга во время запроса

Хотя бустинг во время запроса является гибким, он также имеет некоторые существенные недостатки, влияющие на производительность, масштаб и релевантность запросов, что может сделать его использование в определенных случаях неподходящим:

- он требует дополнительного поиска для поиска бустов перед выполнением поиска с бустингом, что добавляет дополнительную обработку (выполнение двух поисков) и задержку (окончательный запрос должен ждать результатов запроса на поиск сигналов перед обработкой);
- это приводит к неудачным компромиссам между релевантностью (повышение значимости всех релевантных документов) и масштабируемостью (ограничение количества повышаемых документов для поддержания разумного времени запроса и пропускной способности запроса);
- это делает пагинацию¹ неэффективной и потенциально неточной, так как увеличение количества бустов для учета увеличивающегося разнообразия документов при пролистывании результатов замедлит запрос и может привести к перемещению документов на более ранние страницы (пропущенные пользователем) или более поздние страницы (воспринимаемые пользователем как дубликаты).

Первый недостаток прост, так как каждый запрос, по сути, становится двумя запросами, выполняемыми один за другим, что увеличивает общее время поиска. Второй недостаток может быть не таким очевидным. При повышении времени запроса мы ищем определенное количество документов, чтобы подняться выше в результатах поиска по запросу. В нашем примере поиска на iPad из рис. 8.1 (см. листинг 4.7 для кода) буст для запроса в конечном итоге становится

```
"885909457588"^966 "885909457595"^205 "885909471812"^202
"886111287055"^109
"843404073153"^73 "885909457601"^62 "635753493559"^62 "885909472376"^61
"610839379408"^29 "884962753071"^28
```

Этот буст содержит 10 документов, но только потому, что это количество усиления, которые мы запросили. Если предположить, что мы показали только десять документов на первой странице, то вся первая страница будет выглядеть хорошо, но что, если пользователь перейдет на страницу 2? В этом случае не будет показано ни одного документа с повышенным рейтингом, поскольку бустинг был сделан только для первых 10 документов с сигналами для запроса!

Чтобы сделать бустинг для документов второй страницы, нам нужно будет убедиться, что у нас есть по крайней мере достаточно усиления документов, чтобы покрыть полные первые две страницы, что означает увеличение бустинга с 10 до 20 (изменение параметра `limit` до 20 в поисковом запросе повышения):

```
"885909457588"^966 "885909457595"^205 "885909471812"^202
"886111287055"^109 "843404073153"^73 "635753493559"^62 "885909457601"^62
"885909472376"^61 "610839379408"^29 "884962753071"^28 "635753490879"^27
```

¹ Пагинация в программировании – это способ организации большого объема данных на веб-сайте путем разделения их на отдельные страницы для удобства навигации пользователей. – *Прим. ред.*


```
"885909457632"^26 "885909393404"^26 "716829772249"^23 "821793013776"^21  
"027242798236"^15 "600603132827"^14 "886111271283"^14 "722868830062"^13  
"092636260712"^13
```

Вы можете решить эту задачу, увеличив количество просматриваемых бустов каждый раз, когда кто-то переходит на «следующую» страницу, но это агрессивно замедлит последующие запросы, так как страница 3 потребует поиска и применения 30 бустов, страница 10 потребует 100 бустов, и т. д. Для варианта использования, где для каждого запроса существует только небольшое количество усиленных документов, это не большая проблема, но для многих вариантов использования могут быть сотни или тысячи документов, которые выиграют от бустинга. В нашем запросе для *ipad* есть более 200 документов, которые содержат агрегированные сигналы, поэтому большинство из этих документов никогда не будут усилены, если только кто-то не углубится в результаты поиска. В этот момент запросы, скорее всего, будут медленными и даже могут истечь по времени ожидания.

Включение только подмножества бустов представляет собой еще одну проблему: результаты поиска не всегда строго упорядочены по значению усиления! Мы предположили, что запрос 10 лучших бустов будет достаточным для первой страницы из 10 результатов, но усиление – это только один из факторов, влияющих на релевантность. Возможно, документы, находящиеся ниже в списке бустов, имеют более высокий базовый балл релевантности, и, если бы их бусты также были загружены, они бы переместились на первую страницу результатов поиска.

В результате, когда пользователь переходит со страницы 1 на страницу 2 и количество загруженных бустов увеличивается, может произойти нежелательный реранкинг, когда результаты могут перескочить на страницу 1 и никогда не быть увиденными или перескочить на страницу 2 и снова быть увиденными как дубликаты.

Даже если эти результаты намного более релевантны, чем результаты поиска без применения бустинга сигналов, это не обеспечивает оптимального пользовательского опыта. Бустинг сигналов во время индексирования может помочь преодолеть эти недостатки, как мы покажем в следующем разделе.

8.6.2. Реализация бустинга сигналов во время индексирования

Бустинг сигналов во время индексирования переворачивает задачу бустинга с ног на голову – вместо усиления популярных документов для запросов во время запроса мы усиливаем популярные запросы для документов во время индексирования. Это достигается путем добавления популярных запросов в поле в каждом документе вместе с их значением усиления. Затем во время запроса мы просто ищем по новому полю, и, если поле содержит термин из нашего запроса, оно автоматически усиливается на основе значения усиления, проиндексированного для термина.

При реализации бустинга во время индексирования мы используем те же самые агрегации сигналов для генерации пар документов и повышения весов для каждого запроса. После того как эти бусты

сигналов сгенерированы, нам просто нужно добавить один дополнительный шаг в наш рабочий процесс: обновить коллекцию продуктов, чтобы добавить поле в каждый документ, содержащий каждый термин, для которого документ должен быть повышен, вместе с ассоциированными сигналами этого термина бустинга. Следующий листинг демонстрирует этот дополнительный шаг.

Листинг 8.8. Индексирование бустов в основной коллекции продуктов

```
from aips.data_loaders import index_time_boosts
```

```
boosts_collection = engine.get_collection("normalized_signals_boosts")
create_view_from_collection(boosts_collection,
                            boosts_collection.name)
```

Загружает ранее созданную модель бустинга сигналов.

```
boosted_products_collection = \
    engine.get_collection("products_with_signals_boosts")
create_view_from_collection(boosted_products_collection,
                            boosted_products_collection.name)
```

```
boosted_products = index_time_boosts.load_dataframe(
    boosted_products_collection,
    boosts_collection)
```

Регистрирует таблицу продуктов, чтобы мы могли загрузить из нее и сохранить обратно с добавленными бустами.

```
boosted_products_collection.write(boosted_products)
```

Вставляет все ключевые слова с бустами сигналов для каждого документа в новое поле `signals_boosts` в документе.

Сохраняет продукты обратно в коллекцию усиленных продуктов с добавленными обновленными `signals_boosts`.

Код считывает все ранее сгенерированные бусты сигналы для каждого документа продукта, а затем сопоставляет запросы и бусты в новое поле `signals_boosts` в этом документе. Поле `signals_boosts` имеет разделенный запятыми список терминов (запросов пользователей) с соответствующим весом для каждого термина.

При использовании Solr в качестве поисковой системы (по умолчанию) это поле `signals_boosts` является специализированным полем, содержащим фильтр `DelimitedPayloadBoostFilter`, позволяющий индексировать термины (запросы) с ассоциированными бустами, которые можно использовать для влияния на рейтинг результатов поиска во время выполнения. Например, для самого популярного iPad документ о продукте теперь будет выглядеть следующим образом:

```
{...
  "id": "885909457588",
  "name": "Apple® - iPad® 2 с Wi-Fi - 16 ГБ - Black"
  "signals_boosts": "ipad|2939,ipad 2|1104,ipad2|540,i pad|341,apple
ipad|
152,ipads|123,apple|118,i pad 2|99,tablets|67,..."
...
}
```

Этот формат бустинга терминов при создании индекса будет отличаться в разных поисковых системах. Во время запроса будет выполняться поиск по этому полю `signals_boosts`, и, если запрос присутствует в поле, показатель релевантности для этого документа будет повышен индексированной полезной нагрузкой для соответствующего запроса. Следующий листинг демонстрирует, как выполнить запрос с использованием бустинга сигналов во время индексирования.

Листинг 8.9. Ранжирование результатов поиска с помощью бустинга сигналов во время индексирования

Бустинг показателя релевантности на основе индексированных сигналов повышает эффективность запроса.

```
def get_boosted_search_request(query, boost_field):  
    request = product_search_request(query)  
    request["index_time_boost"] = (boost_field, query)  ←  
    return request  
query = "ipad"  
boosted_query = get_boosted_search_request(query, "signals_boosts")  
response = boosted_products_collection.search(**boosted_query)  
display_product_search(query, response["docs"])
```

Хотя поддержка запросов по терминам, усиленным во время индексирования, в разных поисковых системах обрабатывается по-разному, в случае Solr (наша поисковая система по умолчанию) это внутренне преобразуется в значение параметра усиления `payload` ("`signals_boosts`", "`ipad`", `1`, "`first`"), добавляемое к поисковому запросу для бустинга документов с помощью полезной нагрузки, прикрепленной к *первому* совпадению запроса `ipad` в поле `signals_boosts` (или `1`, если полезная нагрузка не была проиндексирована). Смотрите раздел 3.2, если хотите освежить в памяти информацию о влиянии на ранжирование таким образом с помощью функций и мультипликативных бустов.

На рис. 8.5 показаны результаты этого бустинга сигналов во время индексирования. Как вы можете видеть, теперь результаты выглядят похожими на вывод бустинга сигналов во время запроса, показанный ранее на рис. 8.1.

Оценки релевантности, скорее всего, не будут идентичными при бустинге во время индексирования и бустинге во время запроса, поскольку при оценке индексированной полезной нагрузки, по сравнению с усиленным термином запроса, математика отличается. Однако относительный порядок результатов должен быть схожим. Бустинг сигналов во время индексирования также будет применяться ко всем документам с соответствующей полезной нагрузкой усиления сигналов, тогда как их бустинг во время запроса применяется только к первым `N` документам, которые явно усилены в запросе.



Рис. 8.5. Бустинг сигналов во время индексирования, демонстрирующий результаты, схожие с бустингом индекса во время запроса

8.6.3. Компромиссы при реализации бустинга во время индексирования

Бустинг во время индексирования, как мы видели, перемещает большую часть работы по усилению сигналов из фазы выполнения запроса в фазу индексирования ваших поисков. Это решает некоторые проблемы, присущие бустингу во время запроса, но также вносит некоторые новые проблемы, которые мы обсудим в этом разделе.

Преимущества бустинга во время индексирования

Бустинг во время индексирования устраняет большинство недостатков бустинга во время запроса.

- Рабочий процесс запроса проще и быстрее, поскольку не требуется выполнять два запроса – один для поиска бустов сигналов, а другой для запуска усиленного запроса с их использованием.
- Каждый запрос становится эффективнее и быстрее по мере увеличения количества усиленных документов, поскольку запрос представляет собой поиск по одному ключевому слову в поле `signals_boosts`, а не более длинный запрос, содержащий увеличивающийся список усиленных документов.
- Разбиение результатов на страницы¹ больше не является проблемой, поскольку усилены все документы, соответствующие запросу, а не только первые *N*, которые можно эффективно загрузить и добавить в запрос.

Учитывая эти характеристики, бустинг во время индексирования может существенно улучшить релевантность и согласованность упо-

¹ Разбиение результатов на страницы, или пагинация, в программировании – это способ организации большого объема данных на веб-сайте путем разделения их на отдельные страницы для удобства навигации пользователей. – *Прим. ред.*

рядочивания результатов, гарантируя, что все запросы получают согласованное и полное усиление всех соответствующих им документов. Он также может существенно повысить скорость запроса, удаляя термины запроса (бусты документа) и устраняя дополнительные поиски перед выполнением основного запроса.

Недостатки бустинга времени индексации

Если бустинг во время индексации решает все проблемы бустинга во время запроса, почему бы нам не использовать бустинг сигналов во время индексации?

Главный недостаток бустинга во время индексации заключается в том, что, поскольку значения бустинга для запроса индексируются для каждого документа (каждый документ содержит термины, для которых этот документ должен быть усилен), добавление или удаление ключевого слова из модели бустинга сигналов требует переиндексации всех документов, связанных с этим ключевым словом. Если агрегации бустинга сигналов обновляются инкрементно (на основе ключевых слов), это означает потенциальную переиндексацию всех документов в вашей поисковой системе на постоянной основе. Если ваша модель бустинга сигналов обновляется пакетно для всего вашего индекса, это означает потенциальную переиндексацию всех ваших документов каждый раз, когда ваша модель бустинга сигналов регенерируется.

Такого рода давление индексации добавляет эксплуатационную сложность вашей поисковой системе. Чтобы поддерживать высокую и постоянную производительность запросов, учитывая эту нагрузку индексации, вы можете захотеть разделить индексацию документов на отдельные серверы, где размещены поисковые индексы для обслуживания запросов.

Разделение задач: индексация и запросы

При выполнении индексации большого объема архитектурно может быть лучше изолировать ваши серверы индексации от ваших серверов запросов, если это возможно. В противном случае нагрузка на память или ЦП из-за загруженных операций индексации может повлиять на задержку и пропускную способность запросов. Не все поисковые системы поддерживают такое разделение задач между серверами индексации и запросов, но многие поддерживают.

Elasticsearch и OpenSearch, например, поддерживают это разделение задач с помощью понятия *индексов подписчиков* (follower indexes), в то время как Solr делает это посредством поддержки различных типов репликации¹ (replica types). Все три из этих движков имеют понятия шардов² (shards), которые являются разделами, содержащими

¹ Репликация – это процесс, под которым понимается копирование данных из одного источника на другой (или на множество других) и наоборот. При репликации изменения, сделанные в одной копии объекта, могут быть распространены в другие копии. – Прим. ред.

² Каждый шард представляет собой отдельный узел внутри кластера, который может состоять из одной или нескольких реплик – серверов, на которых дублируются данные в рамках шарда. – Прим. ред.

подмножество документов в коллекции, и реплик, которые являются точными копиями всех данных, принадлежащих их шарду. У каждого шарда есть лидер, который отвечает за получение обновлений и пересылку их всем репликам.

По умолчанию лидеры отправляют все обновления документов в каждую реплику шарда, и каждая реплика затем (избыточно) индексирует документ, чтобы он был немедленно доступен для поиска в этой реплике. К сожалению, компромиссом для немедленной доступности является то, что индексация большого объема будет потреблять ресурсы в каждой реплике, что может замедлить производительность запросов во всей поисковой системе.

Изменив тип реплики в Solr для реплик на серверах запросов с NRT (т. е. почти в реальном времени) на `LOG` (журнал транзакций) или `PULL`, вы дадите указание репликам извлекать предварительно созданные индексированные файлы из лидера сегмента (который уже индексирует документы) вместо выполнения дублирующей индексации. Аналогично в Elasticsearch и OpenSearch, если вы настраиваете индекс подписчика, серверы, на которых размещен индекс подписчика, будут реплицировать предварительно созданные файлы индекса из индекса лидера вместо избыточной индексации документа. Некоторые другие поисковые системы и векторные базы данных обладают аналогичными возможностями разделения операций индексации и операций запросов на серверах, которые вы можете изучить.

Если вы планируете выполнять бустинг сигналов во время индексации и ожидаете постоянной повторной индексации сигналов в больших объемах, вам следует настоятельно рассмотреть возможность изоляции индексирования и операций во время запроса. Это гарантирует, что производительность ваших запросов не будет отрицательно влиять на значительные дополнительные накладные расходы индексации от текущих агрегаций бустинга сигналов.

Другим недостатком бустинга во время индексирования является то, что внесение изменений в функцию бустинга сигналов может потребовать больше планирования. Например, если вы хотите изменить вес для сигналов клика или покупки с 1:25 на 1:20, вам нужно будет создать поле `signals_boosts_2` с новыми весами, переиндексировать все ваши документы, добавив новые бусты, а затем перевернуть ваш запрос, чтобы использовать новое поле вместо исходного поля `signals_boosts`. В противном случае ваши значения бустинга и рейтинговые баллы будут колебаться непоследовательно, пока все баллы ваших документов не будут обновлены.

Однако, если эти недостатки можно обойти, реализация бустинга сигналов во время индексирования может решить все недостатки бустинга сигналов во время запроса, что приведет к лучшей производительности запросов, полной поддержке разбиения результатов на страницы и использованию всех сигналов из всех документов, а не только выборки из самых популярных.

Как мы увидели в этой главе, бустинг сигналов позволяет популяризировать релевантность – повышать самые важные предметы для определенных запросов. В следующей главе мы реализуем персонализированную релевантность – корректировку рейтинга для каждого пользователя, используя сигналы каждого пользователя (относительно других пользователей), чтобы узнать их конкретные интересы.

Резюме

- Бустинг сигналов – это тип алгоритма ранжирования, который объединяет количество сигналов пользователя на запрос и использует эти количества в качестве повышения релевантности для этого запроса в будущем. Это гарантирует, что самые популярные предметы для каждого запроса будут перемещены в верхнюю часть результатов поиска.
- Нормирование запросов путем обработки различных вариаций (регистр, написание и т. д.) как одного и того же запроса помогает очистить шум в сигналах пользователя и построить более надежную модель бустинга сигналов.
- Краудсорсинговые данные подвержены манипуляциям, поэтому важно явно предотвращать спам и вредоносные сигналы, влияющие на качество ваших моделей релевантности.
- Вы можете объединить различные типы сигналов в одну модель бустинга сигналов, назначая относительные веса каждому типу и выполняя взвешенную сумму значений по типам сигналов. Это позволяет вам придавать большую релевантность более сильным сигналам (положительным или отрицательным) и уменьшать шум от более слабых сигналов.
- Введение функции временного затухания позволяет недавним сигналам иметь больший вес, чем старые сигналы, позволяя старым сигналам с течением времени постепенно исчезать.
- Модели бустинга сигналов могут быть реализованы с использованием бустинга сигналов во время запроса (более гибкое) или бустинга сигналов во время индексирования (более масштабируемое и более последовательное в плане ранжирования релевантности).

Персонализированный поиск

В этой главе рассматривается:

- спектр персонализации между поиском и рекомендациями;
- реализация совместной фильтрации и персонализации с использованием латентных признаков из сигналов пользователей;
- использование эмбедингов для создания профилей персонализации;
- мультимодальная персонализация из контента и поведения;
- применение защитных барьеров персонализации на основе кластеризации;
- избежание ловушек персонализированного поиска.

Чем лучше ваша поисковая система понимает ваших пользователей, тем больше вероятность, что она сможет успешно интерпретировать их запросы. В главе 1 мы представили три ключевых контекста, необходимых для правильной интерпретации намерения запроса: понимание контента, понимание домена и понимание пользователя. В этой главе мы углубимся в контекст понимания пользователя.

Мы уже сосредоточились на изучении домен-специфичного контекста из документов (глава 5) и на самых популярных результатах, по мнению многих разных пользователей (глава 8), но не всегда разумно предполагать, что «лучший» результат согласован среди всех пользо-

вателей. В то время как модели бустинга сигналов находят *самые популярные* ответы среди всех пользователей, персонализированный поиск вместо этого пытается узнать о *специфических* интересах каждого пользователя и вернуть результаты поиска, отвечающие этим интересам.

Например, при поиске ресторанов, несомненно, имеет значение локация пользователя. При поиске работы может иметь значение история занятости каждого пользователя (предыдущие должности, уровень опыта, диапазон зарплат) и локация. При поиске продуктов могут иметь значение определенные предпочтения в отношении бренда, цвета бытовой техники, приобретенные дополнительные продукты и схожие личные вкусы.

В этой главе мы будем использовать сигналы пользователя для изучения латентных признаков, описывающих интересы пользователей. *Латентные признаки* – это признаки, скрытые в данных, но которые можно вывести о пользователях или предметах путем моделирования данных. Эти латентные признаки будут использоваться для генерации рекомендаций по продуктам и повышения персонализированных результатов поиска. Мы также будем использовать эмбединги на основе контента для соотношения продуктов, также мы будем использовать эмбединги продуктов, с которыми взаимодействует каждый пользователь, для генерации векторных профилей персонализации для персонализации результатов поиска.

Наконец, мы сгруппируем продукты по их эмбедингам для генерации защитных барьеров персонализации, чтобы гарантировать, что пользователи не увидят персонализированные результаты поиска на основе продуктов из несвязанных категорий.

Персонализацию следует применять к результатам поиска очень осторожно. Легко разочаровать пользователей, переопределив их явное намерение (обычно указанное как ключевые слова поиска) предположениями, основанными на их предыдущей поисковой активности. Мы углубимся в нюансы баланса между преимуществами более персонализированного поиска и потенциальным разочарованием пользователя, вызванным тем, что поисковая система слишком старается читать их мысли. Не все поиски должны быть персонализированы, но, когда это сделано хорошо, вы увидите, как это может значительно улучшить опыт поиска.

9.1. Персонализированный поиск или рекомендации

Поисковые системы и рекомендательные системы представляют собой два конца спектра персонализации, который мы представили в главе 1 (см. рис. 1.5). Мы также обсудили измерения намерения пользователя в главе 1 (см. рис. 1.7), отметив, что полное понимание намерения пользователя требует понимания контента, понимания пользователя и понимания домена. Рисунок 9.1 вновь всплывает на поверхность этих двух ментальных моделей.

Хотя поиск по ключевым словам представляет только понимание контента, а совместные рекомендации представляют только понимание пользователя, они оба могут и должны быть объединены, когда это возможно. *Персонализированный поиск* находится на пересечении поиска по ключевым словам и совместных рекомендаций.

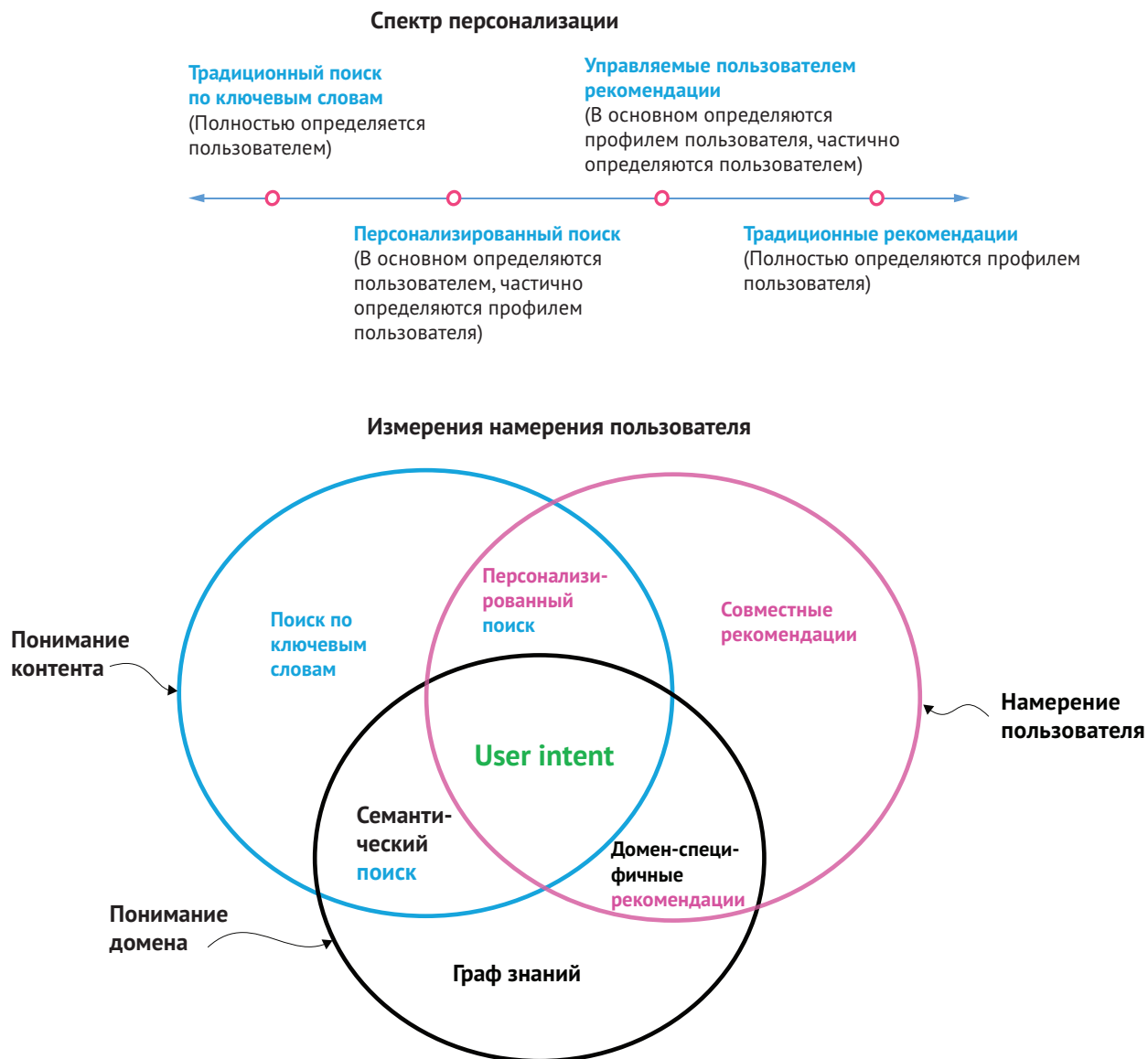


Рис. 9.1. Спектр персонализации и измерения намерения пользователя

Рисунок 9.2 накладывает спектр персонализации поверх диаграммы измерений намерения пользователя, чтобы нарисовать более тонкую картину того, как персонализированный поиск вписывается в спектр персонализации.

Ключевым фактором различия между поисковыми системами и рекомендательными системами является то, что поисковые системы, как правило, руководствуются пользователями и соответствуют их явно введенным запросам, тогда как рекомендательные системы, как правило, не принимают прямого ввода пользователя и вместо этого рекомендуют контент на основе уже известных или предполагаемых зна-

ний. Однако реальность такова, что эти два типа систем образуют две стороны одной медали. Цель в обоих случаях – понять, что ищет пользователь, и предоставить релевантные результаты для удовлетворения информационных потребностей этого пользователя. В этом разделе мы обсуждаем широкий спектр возможностей, лежащих в спектре персонализации между поисковыми и рекомендательными системами.

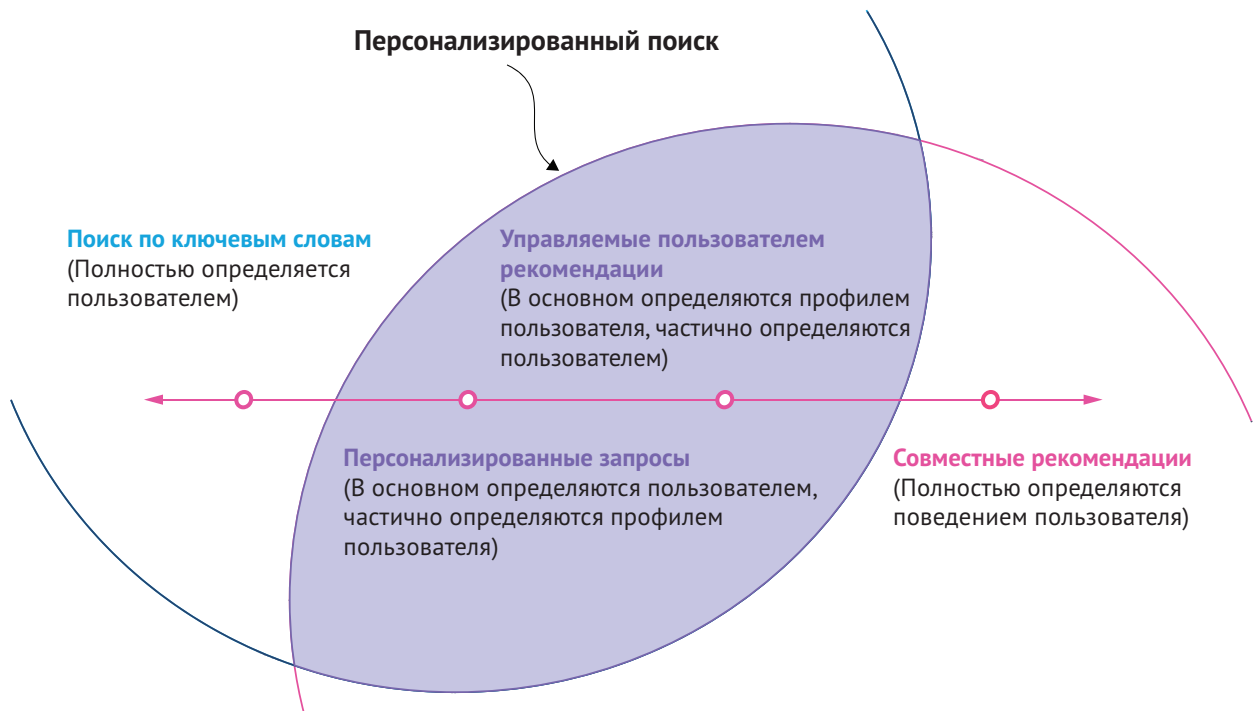


Рис. 9.2. Персонализированный поиск находится на пересечении поиска по ключевым словам и совместных рекомендаций

9.1.1. Персонализированные запросы

Представим, что мы запустили поисковую систему для поиска ресторанов. Наш пользователь Мишель сидит на своем телефоне в Нью-Йорке в обеденное время и вводит в поиске ключевое слово *steamed bageles*. Она видит самые рейтинговые магазины пончиков в Гринвилле, Южная Каролина (США); Колумбусе, Огайо (США); и Лондоне (Великобритания).

Что не так с этими результатами поиска? Что ж, в этом случае ответ очевиден – Мишель ищет обед в Нью-Йорке, но поисковая система показывает ей результаты за сотни или тысячи километров. Но Мишель *никогда не говорила* поисковой системе, что она хочет увидеть результаты только в Нью-Йорке, и не говорила поисковой системе, что она ищет место для обеда поблизости, потому что она хочет поесть прямо сейчас. Тем не менее поисковая система должна иметь возможность вывести эту информацию и персонализировать результаты поиска соответствующим образом.

Рассмотрим другой сценарий – Мишель находится в аэропорту после долгого перелета и ищет на своем телефоне водителя (*driver*). Первые результаты, которые возвращаются, – это клюшка для гольфа для

удара по мячу с ти¹, за которой следует ссылка на драйверы принтера, а затем отвертка. Если поисковая система знает локацию Мишель, разве она не должна сделать вывод о ее предполагаемом значении – что она хочет куда-то поехать?

Используя наш пример поиска работы, приведенный ранее, предположим, что Мишель заходит в свою любимую поисковую систему и вводит `nursing jobs`. Как и в нашем примере с рестораном ранее, разве не было бы идеально, если бы вакансии медсестер в Нью-Йорке отображались вверху списка? Что, если позже она введет `jobs in Seattle`? Разве не было бы идеально, если бы вместо того, чтобы видеть случайные вакансии в Сиэтле (врач, инженер, повар и т. д.), вакансии медсестер теперь отображались вверху списка, поскольку поисковая система ранее узнала, что она медсестра?

Каждый из них является примером персонализированного запроса: объединение как явного запроса пользователя, так и *неявного понимания* намерений и предпочтений пользователя в поиск, который выдает результаты, специально предназначенные для этого пользователя. Сложно сделать такой персонализированный поиск, так как необходимо тщательно сбалансировать свое понимание пользователя, не переопределяя ничего, что он явно хочет запросить. Однако, когда это сделано хорошо, персонализированные запросы могут значительно повысить релевантность поиска.

9.1.2. Рекомендации, управляемые пользователем

Точно так же, как можно добавить неявное понимание атрибутов, специфичных для пользователя, в явный поиск по ключевым словам для генерации персонализированных результатов поиска, также можно включить рекомендации, управляемые пользователем, разрешив пользователю переопределять вводимые данные в автоматически сгенерированных рекомендациях.

Все чаще рекомендательные системы позволяют пользователям видеть и редактировать свои предпочтения в рекомендациях. Эти предпочтения обычно включают список предметов, с которыми пользователь взаимодействовал ранее, просматривая, нажимая или покупая их. В широком спектре вариантов использования эти предпочтения могут включать как конкретные предпочтения по предметам, например любимые фильмы, рестораны или места, так и агрегированные или предполагаемые предпочтения, например размеры одежды, сходство с брендами, любимые цвета, предпочтительные местные магазины, желаемые должности и навыки, предпочтительные диапазоны зарплат и т. д. Эти предпочтения составляют профиль пользователя: они определяют, что известно о клиенте, и чем больше контроля вы можете предоставить пользователю, чтобы он

¹ Ти (tee) – площадка на поле (обычно приподнятая), откуда начинается игра на каждой лунке. Также tee – это маленький колышек, обычно из дерева или пластика, на который гольфист помещает мяч перед первым ударом по лунке. – *Прим. ред.*

мог видеть, настраивать и улучшать этот профиль, тем лучше вы сможете понять своих пользователей и тем счастливее они, вероятно, будут, получив результаты.

9.2. Приближения алгоритмов рекомендаций

В этом разделе мы обсудим различные типы алгоритмов рекомендаций. Реализации механизмов рекомендаций бывают разные, в зависимости от того, какие данные доступны для управления их рекомендациями. Некоторые системы содержат только поведенческие сигналы пользователя и очень мало контента или информации о рекомендуемых предметах, тогда как другие системы имеют богатый контент о предметах, но очень мало взаимодействий пользователя с предметами. Мы рассмотрим рекомендации на основе контента, поведения и мультимодальные рекомендации.

9.2.1. Рекомендации на основе контента

Алгоритмы рекомендаций на основе контента рекомендуют новый контент на основе атрибутов, общих для разных сущностей (часто между пользователями и предметами, между предметами и предметами или между пользователями и пользователями). Например, представьте себе веб-сайт поиска работы. У вакансий могут быть такие свойства, как «должность», «отрасль», «диапазон зарплат», «стаж работы» и «навыки». У пользователей будут похожие атрибуты в их профиле или резюме. Основываясь на этих свойствах, алгоритм рекомендаций на основе контента может определить, какие из этих функций наиболее важны, а затем может ранжировать наиболее подходящие вакансии для любого пользователя на основе желаемых атрибутов пользователя. Это то, что известно как рекомендательная система пользователь–предмет (или от пользователя к предмету).

Аналогично, если пользователю нравится определенная работа, можно использовать этот же процесс для рекомендации похожих работ на основе того, насколько хорошо эти работы соответствуют атрибутам первой работы. Этот тип рекомендаций популярен на страницах с описанием продукта, где пользователь уже просматривает предмет, и может быть желательно помочь ему изучить связанные предметы. Этот тип алгоритма рекомендаций известен как «рекомендатель предмет–предмет».

Рисунок 9.3 демонстрирует, как рекомендатель на основе контента может использовать атрибуты предметов, с которыми пользователь ранее взаимодействовал, чтобы сопоставить похожие предметы для этого пользователя. В этом случае наш пользователь просматривал продукт «моющее средство», а затем ему были рекомендованы «кондиционер для белья» и «листы для сушки» на основе этих предметов, соответствующих одному и тому же полю категории (категория «стирка») и содержащих похожий текст с продуктом «моющее средство» в своих описаниях продуктов. Мы продемонстрировали этот

тип сопоставления связанных атрибутов и категорий в главе 5 при рассмотрении обучения графа знаний и в главе 6 при рассмотрении расширения запроса с использованием графов знаний. В этих случаях мы в основном расширяли запрос по ключевым словам, чтобы включить дополнительные связанные термины, но вы могли сопоставлять предметы на основе любых других атрибутов, таких как бренд, цвет или размер.

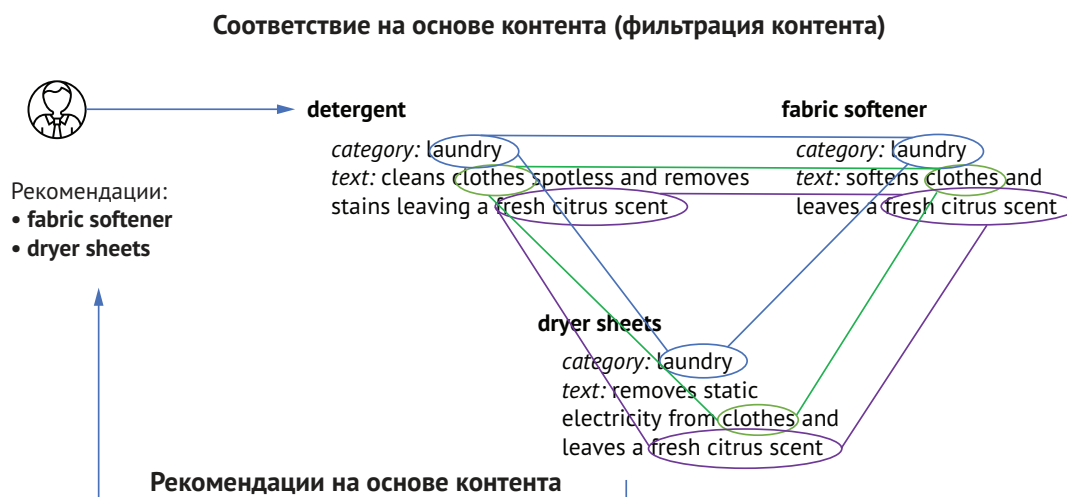


Рис. 9.3. Рекомендации на основе контента, основанные на сопоставлении признаков предмета, представляющего интерес для пользователя, таких как категории и ключевые слова текста

Также можно сопоставлять пользователей с другими пользователями или любую сущность с любой другой сущностью. В контексте рекомендаций на основе контента все рекомендации можно рассматривать как рекомендации по предметам, где каждый предмет является произвольной сущностью, которая разделяет атрибуты с другими рекомендуемыми сущностями.

9.2.2. Рекомендации на основе поведения

Рекомендации на основе поведения используют взаимодействие пользователей с предметами (документами) для обнаружения схожих шаблонов интереса среди групп предметов. Этот процесс называется *совместной фильтрацией*, относящейся к использованию процесса голосования с участием нескольких человек (совместного голосования) для фильтрации совпадений по темам, которые демонстрируют наибольшее сходство, измеряемое тем, сколько похожих по тем или иным признакам пользователей взаимодействовали с одними и теми же предметами. Идея здесь заключается в том, что перекрывающиеся пользователи (т. е. те, у кого схожие предпочтения), как правило, взаимодействуют с одними и теми же предметами, и когда пользователи взаимодействуют с несколькими предметами, они, более вероятно, будут взаимодействовать с похожими предметами, а не с теми, с которыми другие пользователи взаимодействовали меньше.

Одной из удивительных характеристик алгоритмов совместной фильтрации является то, что они процесс оценки релевантности пускают на полный краудсорсинг от ваших конечных пользователей. На самом деле характеристики самих предметов (название, бренд, цвет, текст и т. д.) не нужны – все, что требуется, – это уникальный ID для каждого предмета и знание того, какие пользователи с какими предметами взаимодействовали. Кроме того, чем больше у вас сигналов взаимодействия с пользователем, тем умнее становятся эти алгоритмы, поскольку все больше людей постоянно голосуют и информируют ваш алгоритм ранжирования. Это часто приводит к тому, что алгоритмы совместной фильтрации значительно превосходят алгоритмы, основанные на контенте.

Рисунок 9.4 демонстрирует, как перекрывающиеся поведенческие сигналы от нескольких пользователей могут использоваться для управления совместными рекомендациями. На этом рисунке новый пользователь выражает интерес к удобрению, и поскольку другие пользователи, которые ранее выражали интерес к удобрению, как правило, также кликали, добавляли в корзину или покупали почву и мульчу, то земля или мульча будут возвращены в качестве рекомендаций. Также изображен другой кластер предметов на основе поведения, включая отвертку, молоток и гвозди, но они недостаточно совпадают с текущим интересом пользователя (удобрение), поэтому они не возвращаются в качестве рекомендаций.

Сопоставление на основе поведения (совместная фильтрация)

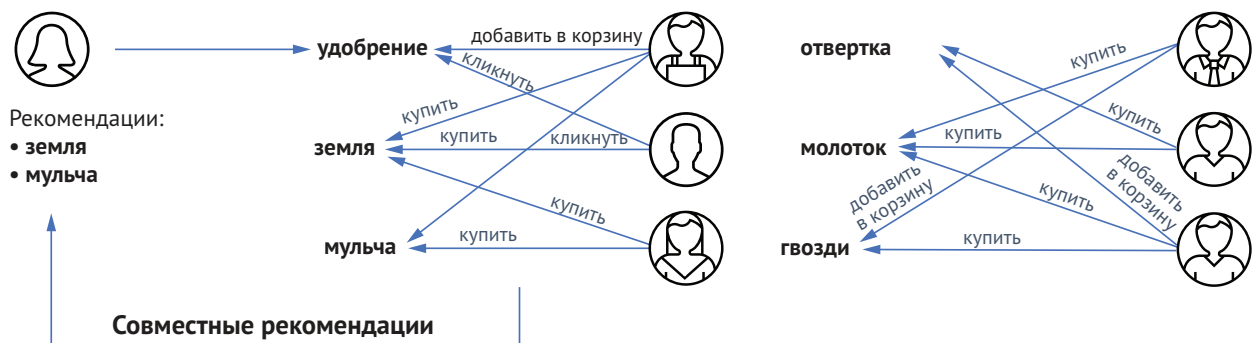


Рис. 9.4. Рекомендации на основе совместной фильтрации, техники, использующей перекрывание поведенческих сигналов нескольких пользователей

Мы даем пример сквозной совместной фильтрации в разделе 9.3, где охватывается процесс обнаружения скрытых признаков пользователя и предмета из поведенческих сигналов пользователя и использование их для создания рекомендаций предметов для пользователей. Поскольку совместная фильтрация полностью краудсорсинговая, она невосприимчива к проблемам качества данных с вашими документами или связанными атрибутами контента, которые могут отсутствовать или быть неверными.

К сожалению, та же зависимость от поведенческих сигналов пользователя, которая делает совместную фильтрацию столь мощной, также оказывается ее слабостью. Что происходит, когда с определенным

предметом происходит всего несколько взаимодействий – или, возможно, их вообще нет? Ответ заключается в том, что предмет либо никогда не будет рекомендован (когда сигналов нет), либо он, скорее всего, будет генерировать плохие рекомендации или отображаться как плохое соответствие другим предметам (когда сигналов мало). Эта ситуация известна как *проблема холодного старта*, и это серьезная проблема для рекомендательных на основе поведения. Чтобы решить эту проблему, вам обычно нужно объединить рекомендательных на основе поведения с рекомендательными на основе контента, как мы обсудим далее.

9.2.3. Мультимодальные рекомендательные системы

Мультимодальные рекомендательные системы (иногда называемые *гибридными рекомендателями*) сочетают в себе как рекомендательные подходы на основе контента, так и подходы на основе поведения. Поскольку совместная фильтрация, как правило, лучше всего работает для предметов с большим количеством сигналов, но плохо работает, когда сигналов мало или нет, часто наиболее эффективно использовать функции на основе контента в качестве базовой линии, а затем накладывать модель совместной фильтрации поверх. Таким образом, если сигналов мало, сопоставитель на основе контента все равно вернет результаты, тогда как если сигналов много, алгоритм совместной фильтрации будет играть большую роль при ранжировании результатов. Объединение может дать вам лучшее из обоих подходов – высококачественное краудсорсинговое сопоставление, избегая при этом проблемы холодного запуска для нового и менее изученного контента. Рисунок 9.5 демонстрирует, как это может работать на практике.

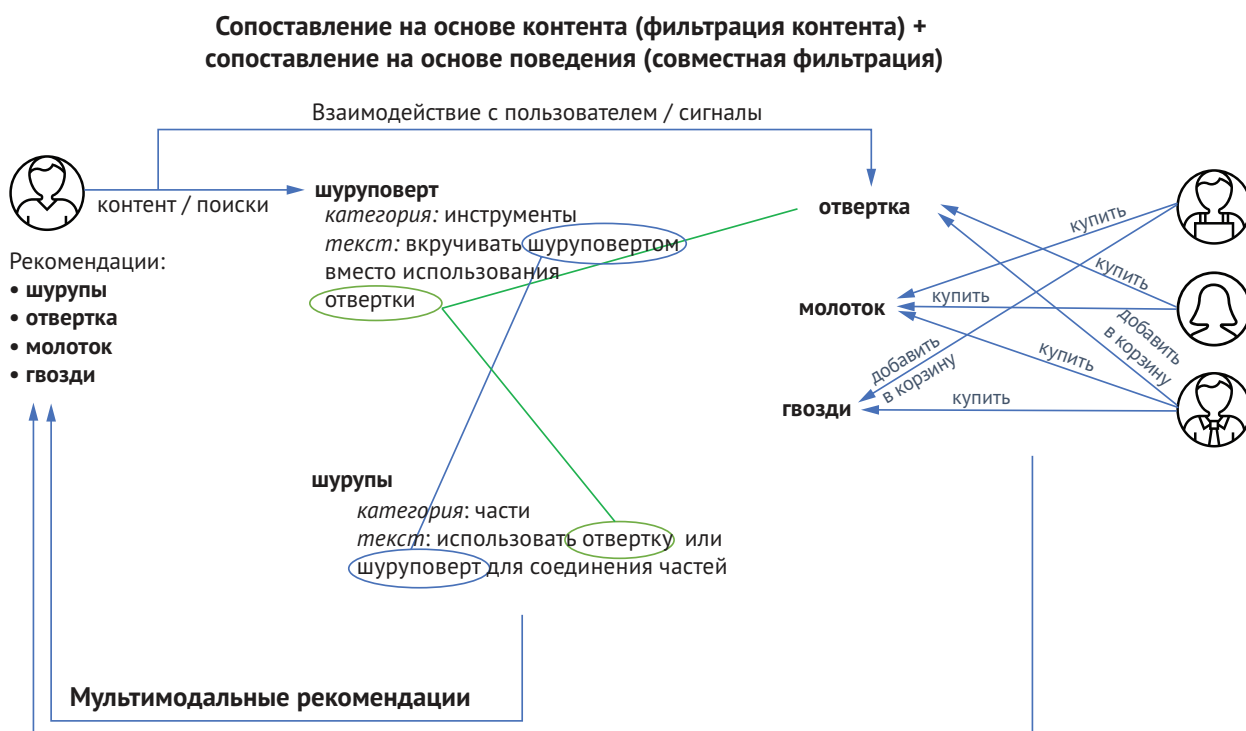


Рис. 9.5. Мультимодальные рекомендации объединяют сопоставление на основе контента и совместную фильтрацию в гибридный алгоритм сопоставления

На рис. 9.5 вы можете видеть, что пользователь может взаимодействовать либо с шурувертом (у которого нет сигналов), либо с отверткой (у которой есть предыдущие сигналы от других пользователей, а также контент), и пользователь будет получать рекомендации в обоих случаях. Это дает преимущество в том, что можно использовать совместную фильтрацию на основе сигналов, а также позволяет сопоставлять на основе контента предметы с недостаточными сигналами.

Мы реализуем модель совместной фильтрации в следующем разделе, а затем гибридную персонализированную систему поиска в разделе 9.4.

9.3. Реализация совместной фильтрации

В этом разделе мы реализуем алгоритм совместной фильтрации. Мы будем использовать сигналы взаимодействия пользователя с предметом и покажем, как изучать скрытые (латентные) признаки из этих сигналов, которые представляют предпочтения пользователей. Затем мы будем использовать эти изученные предпочтения для генерации рекомендаций.

Чистая совместная фильтрация, как на рис. 9.2, позволяет нам изучать сходство между предметами, основываясь исключительно на моделях взаимодействия пользователя с этими предметами. Это мощное понятие, поскольку оно позволяет изучать предметы без каких-либо знаний о самих предметах (например, названия, текст или другие атрибуты).

9.3.1. Изучение скрытых признаков пользователя и предмета с помощью матричной факторизации

Совместная фильтрация часто использует технику, называемую *матричной факторизацией*, для изучения скрытых признаков предметов на основе взаимодействия с пользователем. Скрытые признаки – это характеристики, которые не наблюдаются напрямую, но выводятся из других наблюдаемых признаков. Например, предположим, что у вас есть четыре пользователя со следующей историей покупок фильмов:

- пользователь 1 – «Мстители: Финал», «Черная пантера» и «Черная вдова»;
- пользователь 2 – «Черная вдова», «Капитан Марвел» и «Черная пантера»;
- пользователь 3 – «Черная вдова», «Темный рыцарь» и «Бэтмен»;
- пользователь 4 – «Русалочка», «Король-лев» и «История игрушек»;
- пользователь 5 – «Холодное сердце», «История игрушек» и «Король-лев».

Есть ли закономерности в этих покупках? Если вы знаете названия или описания, вы можете сделать следующие выводы.

Пользователи 1–3:

- все это фильмы о супергероях;
- большинство из них были созданы Marvel Studios, хотя некоторые были созданы Warner Brothers (DC Comics);
- все это боевики;
- они не подходят для маленьких детей из-за насилия или языка.

Пользователи 4–5:

- все это анимационные фильмы;
- все они подходят для маленьких детей;
- все они созданы Disney/Pixar.

Представьте, что у вас нет доступа ни к чему, кроме идентификаторов продуктов. Используя матричную факторизацию, можно наблюдать, как пользователи взаимодействуют с предметами, и делать выводы о скрытых признаках этих предметов. Если признаки, перечисленные в предыдущих пунктах, наиболее прогнозируют покупки похожих пользователей, они, скорее всего, будут представлены в скрытых признаках, полученных с помощью матричной факторизации. Матричная факторизация также, скорее всего, обнаружит другие признаки, которые не столь очевидны.

В качестве другого примера в наборе данных RetroTech сигналы пользователей могут показывать, что одна группа покупает микроволновые печи из нержавеющей стали, холодильники из нержавеющей стали и посудомоечные машины из нержавеющей стали. Другая группа пользователей может покупать черные микроволновые печи, черные холодильники и черные посудомоечные машины. Кластеризуя взаимодействия пользователя с предметами вместе, можно статистически определить скрытый признак, который разделяет эти предметы по цвету. Кроме того, одна группа может покупать телевизоры, PlayStation и DVD-плееры, а другая группа может покупать iPhone, чехлы для телефонов и защитные пленки для экрана. Кластеризуя эти сигналы поведения вместе, мы можем дифференцировать эти категории продуктов (домашние кинотеатры против мобильных телефонов) в один или несколько скрытых признаков.

На рис. 9.6 показан пример матрицы взаимодействия пользователя с продуктами для нескольких продуктов и пользователей. Цифры – это рейтинги, показывающие, насколько сильно пользователь (ось Y) заинтересован в продукте (ось X), при этом покупка имеет больший вес, чем действие добавления в корзину, а сигнал добавления в корзину имеет больший вес, чем клик. Пустые ячейки представляют отсутствие взаимодействия между пользователем и продуктом.

		Предметы					
		Мстители: Финал (фильм)	Черная пантера (фильм)	Ноттинг-Хилл (фильм)	Дневник памяти (фильм)	Майнкрафт (компьютерная игра)	Холостяк (телешоу)
Пользователи		9	7		1	7	
		2		9	10		10
		9	6	3	4	8	1
		10	7	6	8	1	9
		1	3	9		1	7

Рис. 9.6. Матрица взаимодействия пользователя с предметом. Числа представляют предпочтения пользователя относительно предмета по шкале от 1 (очень неблагоприятно) до 10 (очень благоприятно). Пустые ячейки представляют отсутствие взаимодействия между пользователем и предметом

Учитывая матрицу взаимодействия пользователя и предмета, наша цель – выяснить, *почему* определенные предметы предпочитаются каждым из пользователей. Мы предполагаем, что эти предпочтения объясняет некоторая комбинация интересов пользователя и сходства предметов. Таким образом, факторизация матрицы берет матрицу оценок пользователя и предмета и разбивает ее на две отдельные матрицы – одна сопоставляет каждого пользователя с набором признаков, а другая сопоставляет с набором признаков каждый предмет. Рисунок 9.7 демонстрирует процесс факторизации матрицы, в результате которого матрица рейтингов пользователя и предмета R преобразуется в соответствующую матрицу признаков пользователя U и матрицу признаков предмета I .

Каждая строка в матрице пользователя (U) – это вектор, представляющий одного пользователя, каждый столбец которого представляет один из трех скрытых признаков пользователя (обозначенных как Latent User Feature 1, Latent User Feature 2 и Latent User Feature 3). В матрице предметов (I) каждый столбец – это вектор, представляющий один элемент, каждая строка которого представляет один из трех скрытых признаков предмета (обозначенных как Latent User Feature 1, Latent User Feature 2 и Latent User Feature 3).

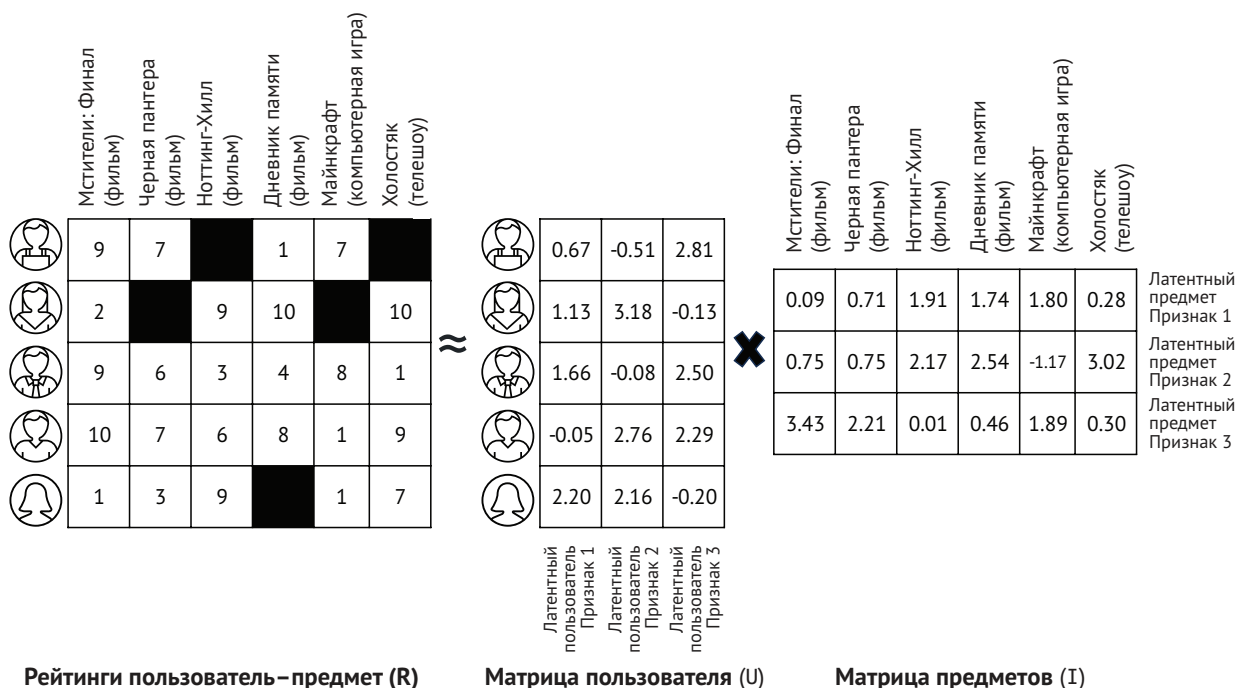


Рис. 9.7. Факторизация матрицы. Матрица пользователь–предмет R разлагается на две матрицы: матрицу пользователя U и матрицу предмета I. Произведение этих двух матриц ($U \cdot I$) должно быть как можно ближе к исходной матрице пользователь–предмет R

У нас нет названий для этих скрытых признаков или мы не знаем точно, что они представляют, но они обнаруживаются математически и предсказывают фактический интерес пользователя к предметам. Количество скрытых признаков – это гиперпараметр, который можно настроить, но в этом примере он равен 3. Это означает, что каждый пользователь представлен вектором с тремя измерениями (скрытые признаки), и каждый предмет также представлен вектором с тремя измерениями (скрытые признаки).

После того как матрицы U и I обучены, их можно использовать независимо друг от друга для прогнозирования сходства между любым пользователем и предметом (путем сравнения пользователей в U с предметами в I), между любыми двумя пользователями (путем сравнения пользователя в U с другим пользователем в U) или между любыми двумя предметами (путем сравнения предмета в I с другим предметом в I). Мы сосредоточимся только на сходстве пользователя и предмета как на средстве персонализации рекомендаций для каждого пользователя. На рис. 9.8 показано, как сгенерировать прогноз рейтинга предмета для любого пользователя.

Для первого пользователя (первая строка в U) мы можем сгенерировать прогнозируемый рейтинг для фильма «Мстители: Финал» (первый столбец в I), выполнив скалярное произведение между первой строкой пользовательской матрицы U (0.67, -0.51, 2.81) и первым столбцом матрицы предметов I (0.09, 0.75, 3.43), что приводит к прогнозируемому рейтингу $(0.67 * 0.09) + (-0.51 * 0.75) + (2.81 * 3.43) = 9.32$. Аналогично для второго пользователя (вторая строка в U) мы можем сгенерировать прогнозируемый рейтинг для фильма «Дневник памяти» (четвертый

	Латентный пользователь Признак 1	Латентный пользователь Признак 2	Латентный пользователь Признак 3		Мстители: Финал (фильм)	Черная пантера (фильм)	Ноттинг-Хилл (фильм)	Дневник памяти (фильм)	Майнкрафт (компьютерная игра)	Холостяк (телешоу)	
	0.67	-0.51	2.81	×	0.09	0.71	1.91	1.74	1.80	0.28	Латентный предмет Признак 1
	1.13	3.18	-0.13		0.75	0.75	2.17	2.54	-1.17	3.02	Латентный предмет Признак 2
	1.66	-0.08	2.50		3.43	2.21	0.01	0.46	1.89	0.30	Латентный предмет Признак 3
	-0.05	2.76	2.29								
	2.20	2.16	-0.20								

Расчет предпочтений из скрытых факторов:

 Мстители: Финал (фильм) $(0.67 * 0.09) + (-0.51 * 0.75) + (2.81 * 3.43) = 9.32$	 Дневник памяти (фильм) $(1.13 * 1.74) + (3.18 * 2.54) + (-0.13 * 0.46) = 9.98$
--	---

Рис. 9.8. Расчет предпочтения пользователь–предмет из факторизованных матриц. Умножьте каждое скрытое значение признака пользователя (первое, второе и третье значения в строке для пользователя) на соответствующее скрытое значение признака предмета (первое, второе и третье значения в столбце для предмета), а затем просуммируйте результаты. Это прогнозируемое предпочтение пользователь–предмет для выбранного пользователя и предмета

столбец в I), выполнив скалярное произведение между второй строкой матрицы пользователя U (1.13, 3.18, -0.13) и четвертым столбцом матрицы предметов I (1.74, 2.54, 0.46), что дает прогнозируемый рейтинг 9.98.

Хотя выполнение индивидуальных прогнозов между одним пользователем и предметом может быть полезным в некоторых случаях, например для генерации рекомендаций в реальном времени сразу после инкрементального взаимодействия пользователя, часто более полезно сгенерировать полную матрицу пользователь–предмет R' прогнозируемых рейтингов для всех пользователей и предметов. На рис. 9.9 показана окончательная матрица пользователь–предмет R' , сгенерированная (справа) путем выполнения скалярного произведения матрицы пользователя U с матрицей предметов I .

При взятии скалярного произведения матрицы пользователя и матрицы предмета ($U \cdot I$) результирующая матрица пользователь–предмет R' должна быть максимально близка к исходной матрице пользователь–предмет R . Минимизация разницы между исходной матрицей R и прогнозируемой матрицей R' является целью оптимизации обучения факторизации матрицы. Чем ближе две матрицы, тем лучше способность модели предсказывать аналогичные персонализированные рекомендации в будущем.

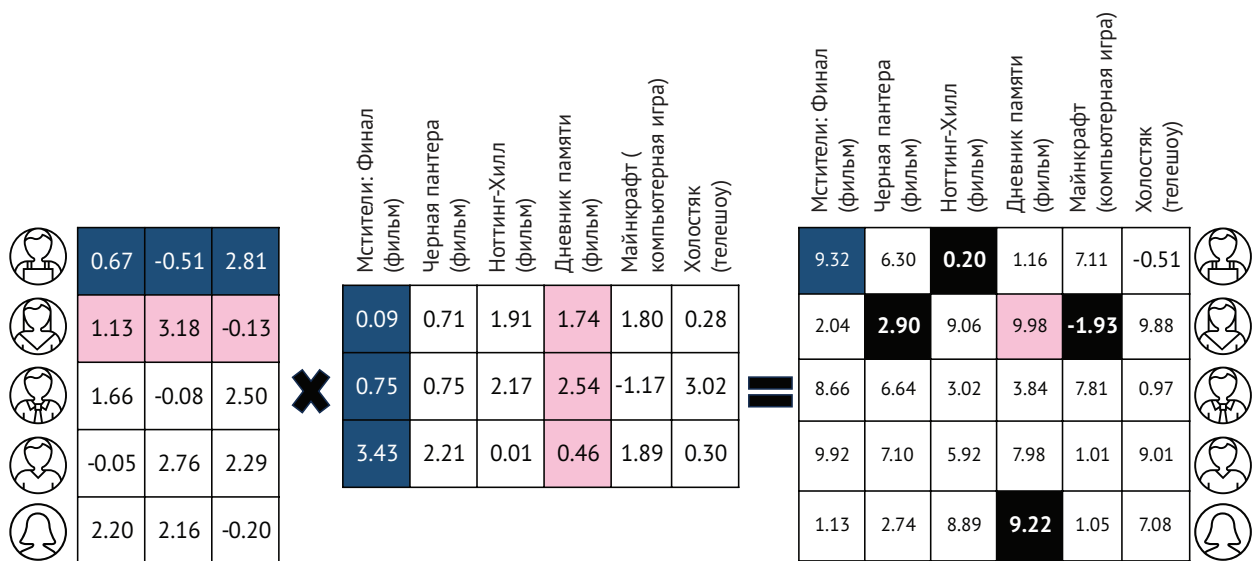


Рис. 9.9. Восстановленная матрица пользователь–предмет R' с выделенными предыдущими вычислениями из рис. 9.8. Обратите внимание, что пустые значения из исходной матрицы пользователь–предмет R теперь заполнены прогнозируемыми значениями (выделено черным)

На практике латентные признаки не идеально представляют все потенциально релевантные признаки. Однако, обучаясь с функцией потерь, которая уменьшает разницу между исходным R и прогнозируемым R' , модель максимизирует шансы представления R и, таким образом, сможет наилучшим образом предсказывать будущие рекомендации на основе прошлых взаимодействий пользователя с предметом.

9.3.2. Реализация совместной фильтрации с помощью метода чередующихся наименьших квадратов

Одним из популярных алгоритмов для чистой совместной фильтрации (основанной только на взаимодействии пользователя с предметами) является *метод чередующихся наименьших квадратов* (ALS). ALS – это итеративный алгоритм, который выполняет факторизацию матрицы, чередуя изучение латентных признаков предметов и латентных признаков пользователей.

Логика ALS заключается в том, что латентные признаки в матрице оценок пользователь–предмет представляют собой комбинацию латентных признаков пользователя и латентных признаков предметов. Хотя относительные веса между пользователями и предметами для каждого скрытого признака заранее неизвестны, можно начать изучать веса признаков пользователя, изначально используя случайные веса предметов и замораживая их (сохраняя их постоянными). По мере того как веса признаков пользователя начинают объединяться, их можно заморозить и использовать в качестве входных данных при изучении весов признаков предмета. Затем ALS продолжает чередовать дальнейшее обучение матрицы пользовательских признаков (с последними замороженными весами признаков предметов) и матрицы признаков предметов (с последними замороженными весами

признаков предметов). Этот процесс повторяется в течение настраиваемого количества итераций, пока веса обеих матриц не будут хорошо сбалансированы и оптимизированы. Чередую обучение латентных признаков предметов и пользователей, ALS может итеративно изучать наилучшие объединенные веса обеих матриц для улучшения предсказательной силы модели.

Количество латентных признаков, изученных с помощью факторизации матрицы, является гиперпараметром, называемым рангом. Чем выше ранг, тем более детализированные признаки вы можете изучить, но вам также, как правило, требуется больше точек данных для надежного изучения более детализированных признаков. Хотя вы не сможете применить метку к каждому скрытому признаку (признаки представлены просто в виде чисел), все равно можно обнаружить значимые категории в данных, которые лучше всего предсказывают похожие предметы. ALS – популярный алгоритм для совместной фильтрации, поскольку его относительно легко реализовать и масштабировать до больших наборов данных.

В этом разделе мы обсудим, как реализовать ALS с помощью Spark для создания модели рекомендаций на основе взаимодействий пользователя и предмета. Мы будем использовать набор данных RetroTech, поскольку он содержит взаимодействия пользователя и предмета для набора продуктов. Мы будем использовать взаимодействия пользователя и предмета для изучения латентных признаков как пользователей, так и предметов, а затем будем использовать эти латентные признаки для создания будущих рекомендаций.

Мы начнем с создания списка неявных предпочтений для каждой пары пользователь–предмет с помощью встроенной реализации ALS Spark. Листинг 9.1 генерирует коллекцию `user_product_implicit_preferences`, назначая рейтинг на основе силы взаимодействия пользователя.

Листинг 9.1. Генерация неявных оценок пользователя и предмета из сигналов пользователя

```
click_weight = 1
add_to_cart_weight = 0
purchase_weight = 0
```

Сейчас взвешиваются только сигналы кликов, но веса можно устанавливать для каждого типа сигнала.

```
signals_collection = engine.get_collection("signals")

mixed_signal_types_aggregation = f"""
SELECT user, product,
       (click_boost + add_to_cart_boost + purchase_boost) AS rating
FROM (
  SELECT user, product,
         SUM(click) AS click_boost,
         SUM(add_to_cart) AS add_to_cart_boost,
         SUM(purchase) AS purchase_boost
  FROM (
    SELECT s.user, s.target AS product,
```

```

    IF(s.type = 'click', {click_weight}, 0) AS click,
    IF(s.type = 'add-to-cart', {add_to_cart_weight}, 0) AS add_to_cart,
    IF(s.type = 'purchase', {purchase_weight}, 0) AS purchase
FROM signals s
WHERE (s.type != 'query')) AS raw_signals
GROUP BY user, product) AS per_type_boosts"""

```

```

signals_agg_collection = \
    aggregate_signals(signals_collection, "user_product_implicit_preferences",
                      mixed_signal_types_aggregation)

```

Объединяет все сигналы для создания единого рейтинга для каждой пары пользователь–предмет.

Мы смоделировали поддержку для сигналов кликов, добавления в корзину и покупки, хотя мы назначили вес 1 только для кликов и 0 для сигналов добавления в корзину и покупки. Мы сделали это, чтобы сделать математику более простой для алгоритма ALS, но вы можете поэкспериментировать с включением сигналов добавления в корзину или покупки, увеличив их вес до положительного числа. Эти веса несколько произвольны, но идея состоит в том, чтобы дифференцировать силу интереса пользователя к продукту на основе уровня его взаимодействия. Вы также можете упростить, просто присвоив рейтинг 1 для каждой пары пользователь–предмет, если у вас нет уверенности, что большее количество взаимодействий пользователя обязательно указывает на более высокий рейтинг или что выбранные вами веса имеют смысл.

Подготовив наши рейтинги пользователь–предмет, мы сгенерируем датафрейм из подготовленной коллекции для обучения и тестирования модели. Наш набор данных содержит менее 50 000 продуктов, и мы будем использовать все из них в листинге 9.2. Однако вы можете изменить количество продуктов в `top_product_count_for_recs` на существенно меньшее число, если хотите быстро его обработать. В зависимости от вашего оборудования и конфигурации ресурсов Docker выполнение кода может занять от нескольких минут до нескольких дней. Для быстрого (но некачественного) выполнения рассмотрите возможность тестирования с 1000 продуктов изначально (`top_product_count_for_recs=1000`), а затем масштабируйте количество по мере необходимости.

Листинг 9.2. Подготовка данных рейтингов пользователь–продукт для обучения

```

create_view_from_collection(signals_agg_collection,
                           "user_product_implicit_preferences")

```

```

top_product_count_for_recs = 50000
user_preference_query = f"""
SELECT user, product, rating

```

Уменьшение количества продуктов может ускорить обучение, но с уменьшением точности.

Возвращает пользователя, продукт и рейтинг.


```

FROM user_product_implicit_preferences
WHERE product IN (
    SELECT product FROM (
        SELECT product, COUNT(user) user_count
        FROM user_product_implicit_preferences
        GROUP BY product
        ORDER BY user_count DESC
        LIMIT {top_product_count_for_recs}
    ) AS top_products)
ORDER BY rating DESC"""

```

Ограничивает количество рекомендаций наиболее популярными продуктами.

```

user_prefs = spark.sql(user_preference_query)

```

Наш датафрейм содержит три столбца: `user`, `product` и `rating`. По соображениям производительности многие алгоритмы машинного обучения (включая реализацию ALS Spark, которую мы будем использовать) предпочитают иметь дело с числовыми идентификаторами вместо строк. Spark содержит вспомогательный объект `StringIndexer`, который можно использовать для преобразования строковых идентификаторов в числовые идентификаторы, и соответствующий объект `IndexToString`, который можно использовать для преобразования числовых идентификаторов обратно в строковые идентификаторы. Листинг 9.3 интегрирует это преобразование идентификаторов в наш датафрейм.

Листинг 9.3. Преобразование идентификаторов в целые числа для алгоритма ALS Spark

```

def order_preferences(prefs):
    return prefs.orderBy(col("userIndex").asc(),
                          col("rating").desc(),
                          col("product").asc())

def strings_to_indexes(ratings, user_indexer,
                      product_indexer):
    transformed = product_indexer.transform(
        user_indexer.transform(ratings))
    return order_preferences(transformed)

def indexes_to_strings(ratings, user_indexer,
                      product_indexer):
    user_converter = IndexToString(inputCol="userIndex",
                                   outputCol="user",
                                   labels=user_indexer.labels)
    product_converter = IndexToString(inputCol="productIndex",
                                      outputCol="product",
                                      labels=product_indexer.labels)
    converted = user_converter.transform(
        product_converter.transform(ratings))
    return order_preferences(converted)

```

Преобразует столбцы пользователя и продукта в столбцы индекса для датафрейма.

Числовые сопоставления индекс–строка для продукта и пользователя.

Выполняет преобразование индекса в строку для идентификатора пользователя.

```

user_indexer = StringIndexer(inputCol="user",
                              outputCol="userIndex").fit(user_prefs)

product_indexer = StringIndexer(inputCol="product",
                                 outputCol="productIndex").fit(user_prefs)

indexed_prefs = strings_to_indexes(user_prefs, user_indexer, product_indexer)
indexed_prefs.show(10)

```

Сопоставляет поле строки пользователя с целочисленным индексом с именем `userIndex`.

Сопоставляет поле строки продукта с целочисленным индексом с именем `productIndex`.

Вывод:

```

+-----+-----+-----+-----+-----+
|  user | product | rating | userIndex | productIndex |
+-----+-----+-----+-----+-----+
|u159789|008888345435|    1 |      0.0 |      5073.0 |
|u159789|014633196870|    1 |      0.0 |      4525.0 |
|u159789|018713571687|    1 |      0.0 |     10355.0 |
|u159789|024543718710|    1 |      0.0 |       263.0 |
|u159789|025192979620|    1 |      0.0 |     12289.0 |
|u159789|025193102324|    1 |      0.0 |     9650.0 |
|u159789|085391163121|    1 |      0.0 |     9196.0 |
|u159789|720616236029|    1 |      0.0 |     2781.0 |
|u159789|801213001996|    1 |      0.0 |    28736.0 |
|u159789|813985010007|    1 |      0.0 |     5819.0 |
+-----+-----+-----+-----+-----+
only showing top 10 rows

```

Как вы можете видеть из листинга 9.3, наш датафрейм теперь содержит два дополнительных столбца: `userIndex` и `productIndex`. Мы будем использовать эти числовые идентификаторы в дальнейшем в коде реализации ALS, прежде чем вызовем функцию `indexes_to_strings` в самом конце для обратного преобразования в наши исходные строковые идентификаторы.

Теперь, когда наш датафрейм предпочтений пользователь–предмет подготовлен, пришло время вызвать алгоритм ALS. ALS запрашивает три параметра: `userCol`, `itemCol` и `ratingCol`, которые соответствуют столбцам `userIndex`, `productIndex` и `rating` в нашем датафрейме. Мы также установим несколько других параметров, включая следующие:

- `maxIter=3` (максимальное количество итераций для запуска);
- `rank=10` (количество латентных признаков для изучения);
- `regParam=0.15` (параметр регуляризации);
- `implicitPrefs=True` (следует ли рассматривать рейтинги как явные или неявные);
- `coldStartStrategy=drop` (как обрабатывать новых пользователей или предметы, которые не присутствовали в обучающих данных).

Листинг 9.4 демонстрирует, как вызвать ALS с этими параметрами.

Листинг 9.4. Обучение модели ALS с использованием Spark

```

from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row

als = ALS (maxIter=3, rank=10, regParam=0.15, implicitPrefs=True,
           userCol="userIndex", itemCol="productIndex",
           ratingCol="rating",
           coldStartStrategy="drop", seed=0)

(training_data, test_data) = \
    user_prefs.randomSplit([0.95, 0.05], 0)

training_data = strings_to_indexes(training_data, user_indexer, product_indexer)
test_data = strings_to_indexes(test_data, user_indexer, product_indexer)

model = als.fit(training_data)
predictions = model.transform(test_data)
evaluator = RegressionEvaluator(metricName="rmse",
                                labelCol="rating",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print(f"Root-mean-square error = {rmse}")

```

**Разделяет предпочтения, 95 %
как обучающие данные и 5 %
как тестовые данные.**

**Обучает модель ALS
с пользовательскими
предпочтениями в об-
учающем наборе.**

**Измеряет обученную модель по пользователь-
ским предпочтениям в тестовом наборе.**

Вывод:

Root-mean-square error = 1.0007877733299877

Теперь вы обучили модель рекомендаций! Мы разделили данные на обучающий набор (95 %) и тестовый набор (5 %), построили модель ALS, а затем запустили оценщик для расчета функции потерь среднеквадратической ошибки (RMSE) для измерения качества модели. RMSE – это мера того, насколько далеки прогнозируемые рейтинги от фактических, поэтому чем ниже RMSE, тем лучше модель. Абсолютное значение RMSE менее важно, чем относительное значение на разных этапах обучения модели, поскольку расчет зависит от масштаба, используемого в базовых данных. Если вы увеличите `maxIter`, найдете оптимальный `rank` и увеличите `top_product_count_for_users` при подготовке данных рейтинга пользователь-продукт, вы, скорее всего, увидите, что RMSE немного уменьшится из-за улучшения модели.

Теперь, когда модель обучена, мы можем использовать ее для генерации рекомендаций. Листинг 9.5 демонстрирует, как генерировать рекомендации по предметам для всех пользователей. Мы сгенерируем 10 рекомендаций для каждого пользователя и отобразим 5 лучших рекомендаций пользователей.

Листинг 9.5. Генерация рекомендаций пользователь–предмет из модели ALS

```
indexed_user_recs = model.recommendForAllUsers(10) \
    .orderBy(col("userIndex").asc())
indexed_user_recs.show(5, truncate=64)
```

Вывод:

```
+-----+-----+
|userIndex|                                     recommendations|
+-----+-----+
|      0|[{6, 0.022541389}, {13, 0.015104328}, {36, 0.010634022}, {20,...|
|      1|[{13, 0.009001873}, {3, 0.007981183}, {23, 0.0050935573}, {31...|
|      2|[{9, 0.06319133}, {17, 0.04681776}, {3, 0.041046627}, {14, 0....|
|      3|[{17, 0.0145240165}, {14, 0.01413305}, {12, 0.012459144}, {39...|
|      4|[{14, 0.006752351}, {4, 0.004651022}, {10, 0.004487163}, {17,...|
+-----+-----+
only showing top 5 rows
```

Обратите внимание, что формат рекомендаций немного неудобен. Мы застряли с `userIndex` вместо нашего исходного `user`, а столбец `recommendations` представляет собой массив структур, каждая из которых содержит `productIndex` и `rating`. Давайте наведем порядок, преобразуя каждую рекомендацию `user-item` в строку и заменяя значения `userIndex` и `productIndex` на наши исходные идентификаторы `user` и `product`. Листинг 9.6 демонстрирует, как это сделать.

Листинг 9.6. Преобразование рекомендаций в окончательный, очищенный формат

```
column_exploder = explode("recommendations").alias("productIndex_rating")
user_item_recs = indexed_user_recs.select("userIndex", column_exploder) \
    .select("userIndex", col("productIndex_rating.*"))
user_item_recs = indexes_to_strings(user_item_recs, user_indexer,
    product_indexer)
user_item_recs = user_item_recs.select("user", "product",
    col("rating").alias("boost"))
```

В этом листинге мы сначала разбиваем рекомендации на отдельные строки для каждой рекомендации со столбцами `res.productIndex` и `res.rating`. После выбора `userIndex` для каждой строки мы выбираем `res.productIndex` как `productIndex` и `res.rating` как `rating`. Наконец, мы преобразуем обратно в `user` и `product` из `userIndex` и `productIndex` и возвращаем `user`, `product` и `boost`.

Давайте сохраним наши рекомендации в коллекцию для будущего использования. Это позволит нам мгновенно обслуживать рекомендации из поисковой системы или использовать их в качестве бустов для персонализации результатов поиска. Листинг 9.7 записывает наш датафрейм рекомендаций пользователь–предмет в коллекцию `user_item_recommendations` в поисковой системе, следуя формату данных, похожему на тот, который мы использовали в главе 8 для представления бустов сигнала.

Листинг 9.7. Индексация рекомендаций в поисковой системе

```
recs_collection = engine.create_collection("user_item_recommendations")
recs_collection.write(user_item_recs)
```

Теперь вы сгенерировали рекомендации по предметам для пользователей на основе их взаимодействия с предметами и сохранили их для будущего использования в коллекции `user_item_recommendations` в поисковой системе. Далее мы покажем, как можно обслуживать эти рекомендации и использовать их для персонализации результатов поиска.

9.3.3. Персонализация результатов поиска с бустингом рекомендаций

Сгенерировав рекомендации по предметам пользователя, мы теперь можем персонализировать результаты поиска. Единственное различие между нашей схемой коллекции для коллекции `signals_boosts` в главе 8 и коллекцией `user_item_recommendations` здесь – это замена столбца `query` на столбец `user`. Другими словами, в то время как бустинг сигналов основан на сопоставлении определенного ключевого запроса и применении связанных с ним бустов релевантности предмета, персонализация основана на сопоставлении определенного пользователя и применении связанных с ним бустов релевантности предмета.

С нашей коллекцией рекомендаций, теперь заполненной из листинга 9.7, мы можем либо обслуживать рекомендации напрямую (без запроса ключевого слова), либо использовать рекомендации для персонализации результатов поиска, усиливая их на основе рекомендаций пользователя.

Чистые совместные рекомендации

Обслуживать рекомендации напрямую просто, поэтому мы начнем с этого. Листинг 9.8 показывает последние сигналы для одного из наших пользователей, для которого мы продемонстрируем эти методы персонализации.

Листинг 9.8. История взаимодействия для нашего целевого пользователя

```
def signals_request(user_id):
    return {"query": "*",
            "return_fields": ["signal_time", "type", "target"],
            "order_by": [("signal_time", "asc")],
            "filters": [("user", user_id)]}

user_id = "u478462"
signals_collection = engine.get_collection("signals")

request = signals_request(user_id)
previous_signals = signals_collection.search(**request)["docs"]
print_interaction_history(user_id, previous_signals)
```

Пользователь, для которого мы будем персонализировать результаты.

Previous Product Interactions for User: u478462

signal_time	type	target	name
05/20 06:05	query	apple	apple
05/20 07:05	click	885909457588 Apple® - iPad® 2 with Wi-Fi - 16GB...	
05/20 07:05	add-to-cart	885909457588 Apple® - iPad® 2 with Wi-Fi - 16GB...	
05/20 07:05	purchase	885909457588 Apple® - iPad® 2 with Wi-Fi - 16GB...	
05/25 06:05	query	macbook	macbook
05/25 07:05	click	885909464043 Apple® - MacBook® Air - Intel® Cor...	

На основе истории пользователя становится ясно, что его интересуют продукты Apple, планшеты и компьютеры. Следующий листинг демонстрирует, как обслуживать рекомендации для этого пользователя из нашей коллекции `user_item_recommendations`.

Листинг 9.9. Обслуживание рекомендаций с использованием запроса бустинга сигналов

```
def get_query_time_boosts(user, boosts_collection):
    request = {"query": "*",
               "return_fields": ["product", "boost"],
               "filters": [("user", user)] if user else [],
               "limit": 10,
               "order_by": [("boost", "desc")]}

    response = boosts_collection.search(**request)
    signals_boosts = response["docs"]
    return " ".join(f'{b["product"]}'^f'{b["boost"] * 100}'
                    for b in signals_boosts)

def search_for_products(query, signals_boosts):
    request = product_search_request(query if query else "*")
    if signals_boosts:
        request["query_boosts"] = ("upc", signals_boosts)
    return products_collection.search(**request)

user = "u478462"
boosts = get_query_time_boosts(user, recs_collection)
response = search_for_products("", boosts)

print(f"Boost Query:\n{boosts}")
display_product_search("", response["docs"])
```

Функция опущена для краткости; ее можно увидеть в листинге 4.3.

Запрашивает коллекцию рекомендаций для индексированных повышений продуктов.

На рис. 9.10 показан вывод листинга 9.9. В верхней части вы увидите список «Запрос бустинга» (Boost Query), показывающий наиболее рекомендуемые продукты для пользователя, а также их относительное повышение для пользователя (которое рассчитывалось как `rating * 100`). Под этим запросом повышения вы увидите результаты поиска с бустингом для этого пустого поиска по ключевым словам, которые являются исходными рекомендациями для пользователя.

Boost Query:
"885909457588"^83.317953 "022265004289"^19.800967 "024543742180"^8.756707 "635753493559"^6.914275
"045496880484"^6.1463382 "635753493573"^5.834811 "885909457595"^5.7118796 "885370315080"^5.6894064
"612572171585"^5.5108927 "885909395095"^5.2595586

Search



Name: Apple® - iPad® 2 with Wi-Fi - 16GB - Black
Manufacturer: Apple®



Name: Samsung - Galaxy Tab 10.1 - 16GB - Metallic Gray
Manufacturer: Samsung



Name: Samsung - Galaxy Tab 10.1 - 32GB - Metallic Gray
Manufacturer: Samsung



Name: Apple® - iPad® 2 with Wi-Fi - 32GB - Black
Manufacturer: Apple®

Рис. 9.10. Рекомендации для пользователя, основанные только на совместной фильтрации

Рекомендации выводят наверх 16-гигабайтный iPad, что имеет смысл, учитывая, что пользователь ранее искал и делал клик на 16-гигабайтный iPad, а другой Apple iPad (модель 32 Гб) занял четвертое место. Вы также видите другие планшеты, произведенные конкурирующими производителями с похожей конфигурацией в верхних рекомендациях. Это хороший пример того, как совместная фильтрация может помочь выявить предметы, которые могут не соответствовать напрямую предыдущим взаимодействиям пользователя (только с ноутбуком Apple и iPad), но которые все еще могут быть релевантны интересам пользователя (аналогичные планшеты iPad).

Подобные рекомендации могут быть полезны для интеграции с традиционными результатами поиска или, возможно, даже для вставки в набор результатов поиска. Но их также можно использовать в качестве бустов для вашего алгоритма ранжирования ключевых слов для персонализации результатов поиска, что мы рассмотрим далее.

Чистый поиск по ключевым словам и персонализированный поиск

Вместо того чтобы предоставлять рекомендации независимо от поиска по ключевым словам, может быть полезно смешивать их в качестве дополнительных сигналов в вашем алгоритме ранжирования

поиска для персонализации результатов. Возвращаясь к нашему последнему примеру, представьте, что наш пользователь, который интересуется iPad и MacBook от Apple, выполняет поиск по ключевым словам для планшета. Как это будет выглядеть иначе, чем если бы рекомендации для планшета использовались для персонализации результатов поиска? Листинг 9.10 запускает запрос до и после применения бустинга сигналов на основе персонализированных рекомендаций пользователя.

Листинг 9.10. Результаты неперсонализированного поиска в сравнении с персонализированным

```
query = "tablet"
response = search_for_products(query, None)
print(f"Non-personalized Query")
display_product_search(query, response["docs"])

response = search_for_products(query, boosts)
print(f"Personalized Query")
display_product_search(query, response["docs"])
```

← Неперсонализированные результаты поиска (поиск только по ключевым словам).

← Персонализированные результаты поиска (ключевое слово + бустинг рекомендаций пользователя).

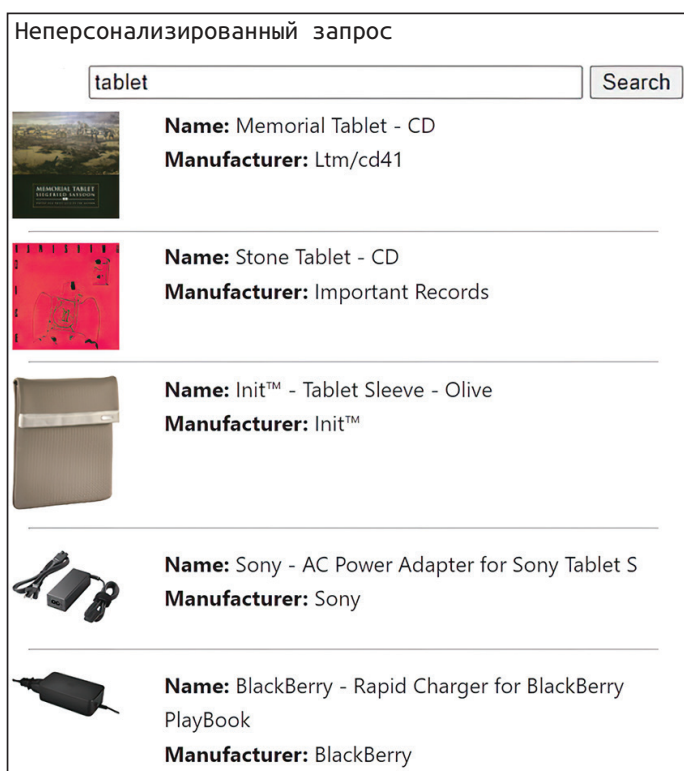


Рис. 9.11. Традиционный поиск по ключевым словам для планшета без применения персонализации

Также обратите внимание, что после усиленных рекомендаций (планшеты из примера рекомендаций) пятый результат поиска – это предмет из неперсонализированных результатов поиска, «Мемориальная доска» под названием CD.

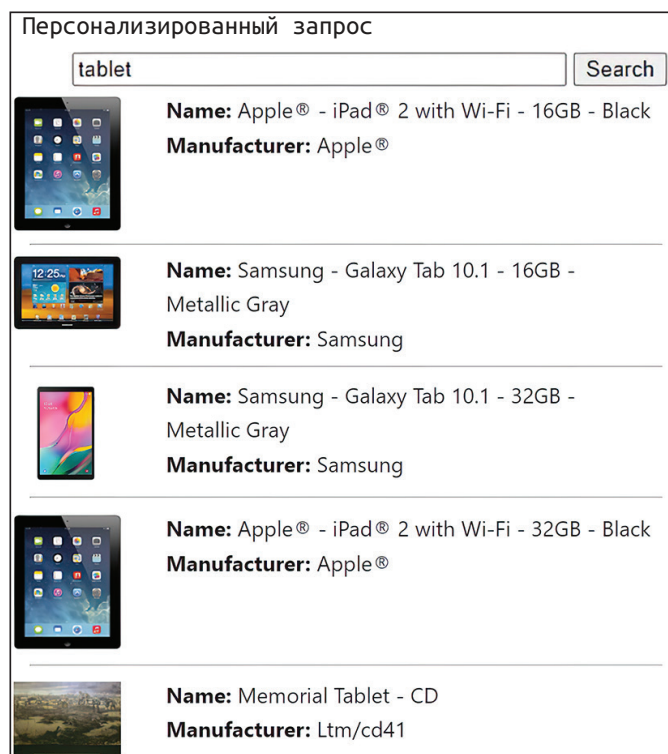


Рис. 9.12. Персонализированный поиск для планшета, где пользователь проявил интерес к бренду Apple

Это подразумевает две вещи:

- если вы персонализировали результаты поиска, а не просто предоставляете чистые рекомендации, вы, вероятно, захотите сгенерировать более 10 рекомендаций на пользователя, особенно потому, что рекомендации отображаются только в том случае, если они также соответствуют явному запросу пользователя;
- алгоритм неперсонализированной релевантности по-прежнему имеет решающее значение. Если бустинг сигналов на основе запроса (согласно главе 8) применялся в дополнение к бустингу рекомендаций (на основе пользователя), вы увидите популярные планшеты вверху (а не чехол планшета и CD), а персонализированные планшеты затем поднимутся выше среди популярных результатов из-за персонализации.

Теперь мы узнали, как работает совместная фильтрация с помощью матричной факторизации, внедрили рекомендации на основе алгоритма совместной фильтрации (ALS) и продемонстрировали, как использовать эти рекомендации для персонализации результатов поиска. В следующем разделе мы рассмотрим еще один метод персонализации на основе эмбедингов документов.

9.4. Персонализация поиска с использованием эмбедингов на основе контента

В предыдущем разделе мы использовали пользовательские сигналы для изучения персонализированных бустов для определенных пред-

метов. Эти усиления были получены путем изучения латентных признаков о пользователях и предметах с использованием матричной факторизации на моделях взаимодействия пользователя с предметом.

Вы также можете использовать эти скрытые факторы напрямую для кластеризации пользователей или предметов вместе. К сожалению, не существует надежного способа сопоставить запросы с определенными кластерами предметов только на основе сигналов взаимодействия пользователя, не увидев соответствующие запросы ранее (снова проблема холодного старта). К счастью, поисковая система довольно редко *не имеет* дополнительных знаний о таких предметах, как заголовки, описания и другие атрибуты.

В этом разделе мы рассмотрим гибридный подход, использующий как понимание на основе контента, так и шаблоны взаимодействия с пользователем для создания развивающегося профиля пользователя для персонализации результатов поиска.

9.4.1. Генерация латентных признаков на основе контента

Мы рассмотрели множество методов использования полей для фильтрации и усиления явных атрибутов в документах. Главы 5–7, в частности, были сосредоточены на создании графов знаний и анализе домен-специфичных сущностей для помощи в обеспечении домен-специфичной релевантности. Хотя эти методы, безусловно, могут быть полезны для реализации персонализированного поиска (и мы призываем вас поэкспериментировать с ними), в этом разделе мы рассмотрим другой подход. Вместо использования явных признаков мы собираемся использовать латентные признаки, полученные из содержимого документов, для создания персонализированных результатов поиска. Мы будем использовать большую языковую модель (LLM) для генерации эмбеддингов для каждого документа, а затем использовать эти эмбеддинги вместе с пользовательскими взаимодействиями наряду с документами для создания развивающегося профиля пользователя. Наконец, мы будем использовать этот профиль пользователя для персонализации результатов поиска.

Рисунок 9.13 демонстрирует, как работает LLM при генерации эмбеддингов для документов. Подобно тому как мы использовали факторизацию матрицы в разделе 9.3.1 для создания матрицы, которая сопоставляла каждый предмет со списком латентных признаков, мы будем использовать LLM для генерации вектора латентных признаков для каждого документа. Мы извлечем эти латентные признаки на основе текста документа, преобразованного в векторное пространство, которое уже было изучено LLM. Не беспокойтесь пока о механике обучения LLM – мы подробно рассмотрим это в главах 14 и 15. Просто знайте, что она обучается на большом корпусе текста и учится сопоставлять слова и фразы в векторном пространстве, используя некоторое количество латентных признаков, которые представляют смысл текста. Каждое из измерений в векторном пространстве представляет собой латентный признак, а значение каждого измерения показывает, насколько сильно этот латентный признак представлен в тексте.



Рис. 9.13. Эмбединги предметов из LLM. Каждое измерение в векторном пространстве представляет собой латентный признак, а значение каждого измерения показывает, насколько сильно этот латентный признак представлен в тексте для этого предмета

Значения на рис. 9.13 приведены для иллюстрации и не являются фактическими значениями, которые будут сгенерированы нашей LLM. Мы присвоили признакам упрощенные метки, чтобы описать то, что они, по-видимому, представляют («размер», «цвет», «похожий на компьютер» и «цена»), но в реальном сценарии эти признаки не будут маркированы и будут представлять более сложные латентные признаки, объединяющие множество различных аспектов, проанализированных глубокой нейронной сетью LLM в процессе обучения.

Для наших примеров мы будем использовать LLM `all-mpnet-base-v2`, общедоступную модель (<https://huggingface.co/sentence-transformers/all-mpnet-base-v2>), которая служит хорошей универсальной LLM для семантического поиска и кластеризации по предложениям и коротким абзацам, как в нашем наборе данных RetroTech. Это легкая модель (всего 768 измерений), которая была обучена на более чем 1.17 млрд пар предложений со всего интернета, что обеспечивает хорошую базу общих знаний.

В следующем листинге извлекаются поля, которые нам нужно передать в LLM.

Листинг 9.11. Извлечение данных о продукте для генерации эмбедингов

```
query = "SELECT DISTINCT name, string(upc), short_description FROM products"
spark.sql(query).createOrReplaceTempView("products_samples")
product_names = dataframe.select("name").rdd.flatMap(lambda x: x).collect()
product_ids = dataframe.select("upc").rdd.flatMap(lambda x: x).collect()
```

Чтобы сгенерировать эмбединги, мы сначала используем Spark для создания новой таблицы `products_samples`, содержащей подмножество полей, полезных для генерации эмбедингов и идентификации связанных

продуктов. Листинг 9.12 демонстрирует, как мы можем генерировать эмбединги для каждого продукта, используя модель LLM `all-mpnet-base-v2` и библиотеку `SentenceTransformers`. Мы сгенерируем объект `product_embeddings`, содержащий вектор размерности 768 для каждого продукта, а также объект `product_names`, содержащий название каждого продукта, и объект `product_ids`, содержащий идентификатор каждого продукта.

Листинг 9.12. Генерация эмбедингов продукта

```
from sentence_transformers import SentenceTransformer
transformer = SentenceTransformer("all-mpnet-base-v2")
```

**Загружает модель
LLM all-mpnet-base-v2.**

...

```
def get_embeddings(texts, model, cache_name, ignore_cache=False):
```

...

```
    embeddings = model.encode(texts)
```

...

```
    return embeddings
```

```
product_embeddings = get_embeddings(product_names,
    transformer, cache_name="all_product_embeddings")
```

**Генерирует векторный эмбединг
размерности 768 для всех продуктов.**

**Код оптимизации для
кеширования сгенери-
рованных эмбедингов
опущен для краткости.**

Поскольку мы используем готовую модель `all-mpnet-base-v2`, загрузка и генерация эмбедингов для всех наших продуктов так же просты, как код в листинге 9.12. Поскольку процесс генерации эмбедингов для всех продуктов может занять некоторое время, блокноты дополнительно содержат некоторые пропущенные оптимизации кода для кеширования и повторного использования эмбедингов, чтобы сэкономить время обработки.

Если мы хотим сравнить два продукта, можем напрямую сравнить их векторы, используя вычисления скалярного произведения или косинусного сходства. 768 признаков в векторе являются предварительно обученными скрытыми признаками каждого документа, аналогичными скрытым признакам, представленным в матрице признаков предмета на рис. 9.7. Это означает, что теперь мы можем:

- генерировать эмбединги для любого предмета или запроса, чтобы получить векторное представление этого предмета или запроса;
- выполнять семантический поиск, начиная с любого эмбединга запроса, и находить другие ближайшие эмбединги (скалярное произведение);
- использовать эмбединг предмета для генерации рекомендаций для других предметов, находя те, у которых эмбединги наиболее схожи (косинус или скалярное произведение).

Но как насчет генерации рекомендаций на основе пользователей или персонализированных результатов поиска? На рис. 9.7 мы не только вынесли за скобки скрытые признаки предмета, но и вынесли за скобки скрытые признаки пользователя. Вся идея совместной фильтрации в разделе 9.2 заключается в том, что похожие пользова-

тели взаимодействуют с похожими предметами именно потому, что предметы имеют общие признаки, которые совпадают с интересами этих пользователей. Другими словами, вектор, представляющий интересы пользователя, должен быть похож на векторы, представляющие предметы, к которым пользователь проявил интерес.

Чтобы персонализировать результаты поиска на основе векторов эмбединга, нам необходимо сгенерировать вектор, представляющий интересы пользователя. Один из способов сделать это – взять среднее значение векторов, представляющих предметы, с которыми взаимодействовал пользователь. Это простой способ создания вектора, представляющего *все прошлые интересы* пользователя, и он работает на удивление хорошо на практике. К сожалению, персонализация каждого будущего поиска на основе *каждого* прошедшего поиска может быть слишком агрессивной, поскольку пользователи часто выполняют несвязанные поиски для разных типов предметов в разное время. Чтобы избежать бесполезной чрезмерной персонализации в этих случаях, может быть полезно сначала применить некоторые ограничения для разных категорий предметов, о чем мы поговорим далее.

9.4.2. Реализация категориальных ограничений для персонализации

Тот факт, что кто-то ищет предмет, не всегда означает, что он хочет видеть похожие предметы. Но обычно очень плохая идея применять персонализацию через понятийные или категориальные границы. Например, если кто-то смотрит фильм «Терминатор», в котором есть жестокие роботы, путешествующие во времени, это не значит, что он хочет купить робот-пылесос или пистолет. В качестве конкретного примера из нашего набора данных представьте, что кто-то ранее выразил интерес к «бутылке для воды Hello Kitty», «электробритве GE (черная)», «лампочкам GE Bright White» и «холодильнику Samsung из нержавеющей стали». Если он впоследствии выполнит поиск по запросу *microwave* (микроволновая печь), какой из предметов на рис. 9.14 будет наиболее подходящим для рекомендации?

Хотя пользователь ранее смотрел на «белые» лампы и «черную» электробритву, нет никаких веских причин применять эти цветовые предпочтения к несвязанной категории «кухонная техника». Кроме того, сомнительно, перейдет ли интерес к «бутылке для воды Hello Kitty» в интерес к «микроволновой печи Hello Kitty» или просмотр «лампочек» и «электробритвы», произведенных компанией «GE», каким-либо образом перейдет в то, что пользователь будет иметь близость к бренду «GE» при просмотре «кухонной техники». Однако, учитывая, что этот конкретный пользователь уже проявил интерес к другому прибору (холодильнику из «нержавеющей стали»), произведенному компанией «Samsung», вполне разумно предположить, что его больше заинтересуют другие приборы из «нержавеющей стали», произведенные «Samsung» (или, по крайней мере, другими компаниями, помимо «GE»), например микроволновая печь, которую он сейчас ищет.

Предыдущие
взаимодействия:

Hello Kitty Water bottle

GE Electric Razor (Black)

GE Bright White Light Bulbs

Samsung Stainless-steel Refrigerator

Нет ограничений персонализации

microwave

Search



Name: Hello Kitty - 0.7 Cu. Ft. Compact Microwave

Manufacturer: Hello Kitty



Name: Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - Stainless-Steel

Manufacturer: Panasonic



Name: Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - Stainless-Steel

Manufacturer: Panasonic



Name: Panasonic - 1.2 Cu. Ft. Mid-Size Microwave - White

Manufacturer: Panasonic

VS.

С категориальными ограничениями персонализации

microwave

Search



Name: Samsung - 1.8 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer: Samsung



Name: Samsung - 1.8 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer: Samsung



Name: Samsung - 1.6 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer: Samsung



Name: Samsung - 1.7 Cu. Ft. Over-the-Range Microwave - Stainless-Steel

Manufacturer: Samsung

Рис. 9.14. Ограничения (guardrails) персонализации могут помочь предотвратить неожиданное влияние несвязанных прошлых интересов на будущие поиски

Персонализацию следует применять с осторожностью. Легко ошибиться и применить персонализацию способом, который не будет полезен пользователю (или даже будет раздражающим и контрпродуктивным), поэтому обычно лучше проявить осторожность и убедиться, что персонализация применяется только тогда, когда она, скорее всего, будет полезна. Один из простых способов сделать это – применять персонализацию только в категориях, похожих на запрос. Это один из способов применения *ограничений*¹ персонализации, и это очень эффективный способ избежать применения персонализации способом, который, скорее всего, для пользователя будет бесполезен.

Хотя ваши данные могут иметь или не иметь явное поле категории для фильтрации, также возможно динамически генерировать категории, кластеризуя предметы вместе на основе их сходства. Это можно сделать, взяв эмбединги для всех предметов и кластеризовав их для динамического создания набора категорий, управляемого данными. Следующий листинг демонстрирует простой метод генерации кластеров предметов из их эмбедингов.

¹ Ограничения (англ. *guardrails*) в программировании – это набор правил, стандартов и лучших практик, связанных с конвейером разработки, от кодирования и построения до тестирования и выпуска. Они позволяют ограничить поведение разработчиков и менеджеров, создать среду, в которой они могут принимать решения независимо. Это помогает снизить риски, уменьшить количество ошибок и сократить время на реагирование на конкретную ситуацию. Также *guardrails* используются для обеспечения надежности и безопасности приложений с использованием языковых моделей. Они позволяют контролировать вывод моделей в определенном формате или контексте, проверяя каждый ответ. – *Прим. ред.*

Листинг 9.13. Генерация динамических категорий из кластеризованных продуктов

```
def get_clusters(data, algorithm, args):
    return algorithm(**args).fit(data)

def assign_clusters(labels, product_names):
    clusters = defaultdict(lambda: [], {})
    for i in range(len(labels)):
        clusters[labels[i]].append(product_names[i])
    return clusters

args = {"n_clusters": 100, "n_init": 10, "random_state": 0}
algo = get_clusters(product_embeddings, cluster.KMeans, args)
labels = algo.predict(product_embeddings)
clusters = assign_clusters(labels, product_names)
```

Генерирует 100 кластеров с использованием алгоритма кластеризации Kmeans.

Назначает каждому названию продукта соответствующую метку кластера.

Чтобы убедиться, что наша кластеризация работает хорошо, мы можем проверить верхние слова в каждом кластере, чтобы убедиться, что они связаны и образуют согласованную категорию. Листинг 9.14 демонстрирует код для определения верхних слов в каждом кластере и для создания 2D-визуализации кластеров с использованием анализа главных компонент (PCA) для преобразования 768-мерных эмбедингов в два измерения для целей визуализации.

Листинг 9.14. Изучение популярных терминов в каждом кластере продуктов

```
import collections, numpy as np, matplotlib.pyplot as plt
from adjustText import adjust_text
from sklearn.decomposition import PCA

plt.figure(figsize=(15, 15))
pca = PCA(100, svd_solver="full")
centers = algo.cluster_centers_
plot_data = pca.fit_transform(centers)

points = []
for i, cluster_name in enumerate(plot_data):
    plt.scatter(plot_data[i,0], plot_data[i, 1],
                s=30, color="k")
    label = f"{i}_{"".join(top_words(clusters[i], 2))}"
    points.append(plt.text(plot_data[i, 0],
                           plot_data[i, 1],
                           label, size=12))
adjust_text(points, arrowprops=dict(arrowstyle="-",
                                     color="gray", alpha=0.3))
plt.show()
```

Выполняет PCA для сокращения эмбедингов до двух измерений для визуализации.

Функция верхних слов получает самые распространенные слова из кластера.

Проходит по каждому кластеру и отображает их на графике.

Добавляет текстовую метку для каждого кластера с ID кластера и топ-N словами в каждом кластере.

Улучшение преобразования: настраивает текстовые метки для минимизации перекрытия.

На рис. 9.15 показан вывод листинга 9.14. Каждая точка представляет кластер, при этом текстовая метка для каждого кластера включает идентификатор кластера и верхние слова в этом кластере.

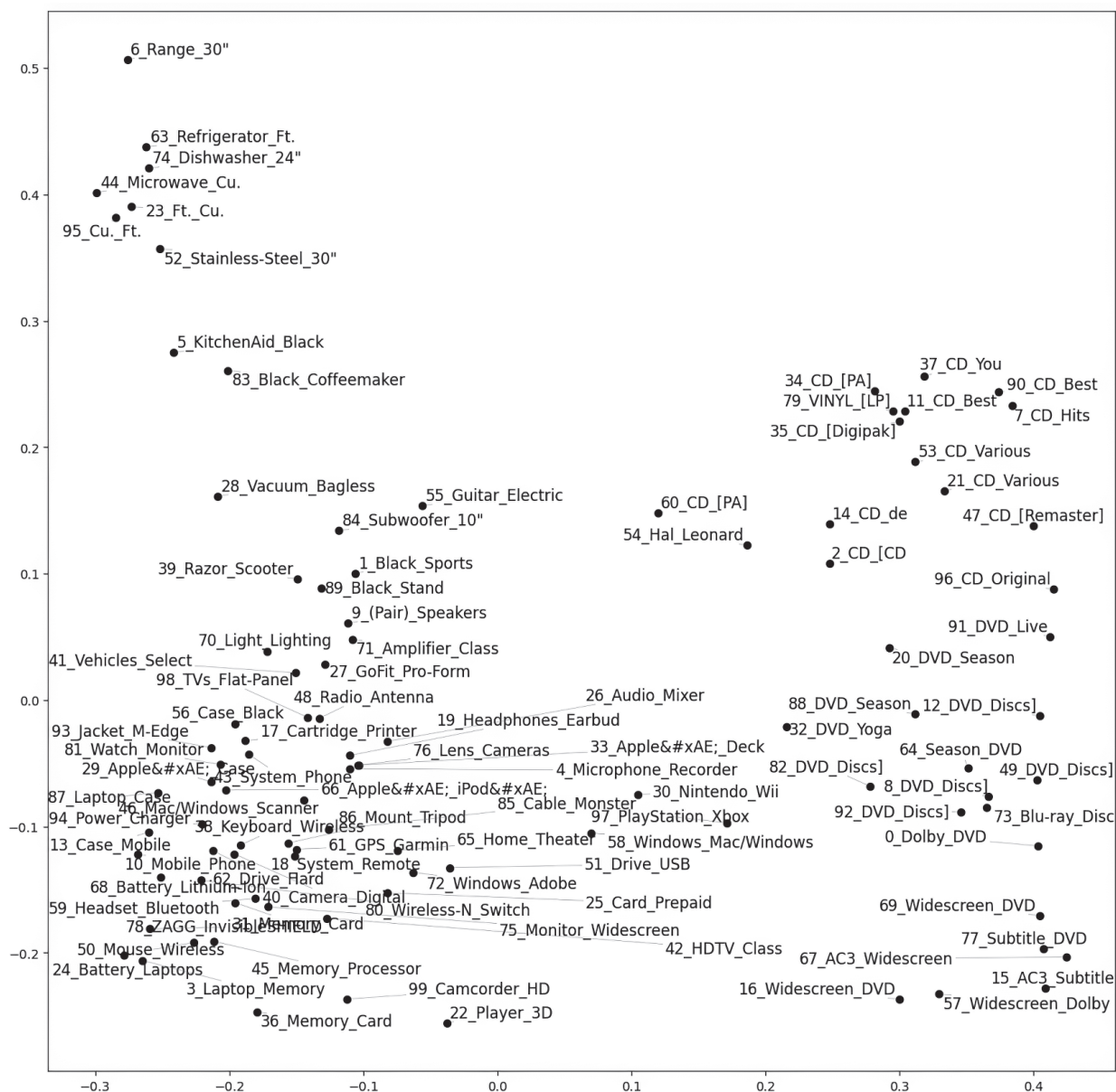


Рис. 9.15. Кластеры, созданные с помощью кластеризации KMeans¹ всех эмбедингов продуктов, которые будут использоваться для категоризации всех запросов и продуктов

Хотя рис. 9.15 может показаться хаотичным, представляя все 100 кластеров, в которые категоризируются наши почти 50 000 продуктов, вы

¹ KMeans – один из самых популярных алгоритмов кластеризации. Алгоритм работает следующим образом. 1. Инициализация. Случайным образом выбираются k центроидов (центров кластеров). 2. Назначение кластеров. Каждый объект данных назначается к ближайшему центроиду. Это делается путем вычисления расстояния от каждого объекта до всех центроидов и выбора минимального расстояния. 3. Обновление центроидов. Центроиды пересчитываются как среднее значение всех объектов, принадлежащих к кластеру. 4. Повторение. Шаги 2 и 3 повторяются до тех пор, пока центроиды не перестанут изменяться или не будет достигнуто максимальное количество итераций. – Прим. ред.

можете увидеть четкие закономерности в семантическом пространстве. Верхний левый угол графика содержит кухонные приборы, а музыка, как правило, находится в правом верхнем углу оставшейся заполненной области графика (CD-диски в правом верхнем углу, музыкальные инструменты и колонки в верхнем среднем углу), а предметы, связанные с видео и хранением данных, как правило, находятся в нижней части графика (DVD и Blu-ray в правом нижнем углу, домашние кинотеатры и камеры в нижней средней части, карты памяти компьютера и хранилища в нижнем левом углу вместе с другими периферийными компьютерными устройствами). Не стесняйтесь проверять различные категории и отношения между кластерами, но осознайте, что они были преобразованы из 768 измерений в 2 измерения, поэтому большая часть богатства, представленного алгоритмом кластеризации KMeans, будет потеряна при визуализации. Теперь, когда у нас есть кластеры, доступные для категоризации продуктов (и сигналы, соответствующие взаимодействующим продуктам), нам нужно убедиться, что мы можем сопоставлять запросы с правильными кластерами. Есть несколько способов сделать это:

- управляемый моделью – просто пропустите запрос через LLM и используйте полученный вектор эмбединга, чтобы найти ближайшие категории;
- управляемый поведением – используйте сигналы запроса и соответствующие сигналы взаимодействия (например, клики), чтобы определить наиболее любимые пользователями категории для популярных запросов;
- управляемый контентом – выполните поиск по ключевым словам или семантический поиск и найдите топовые категории в результатах;
- гибридный – используйте любую комбинацию этих подходов.

Подход, основанный на поведении, следует методологии бустинга сигналов из главы 8, но он объединяет категории, связанные с наиболее усиленными документами, а не запросы. Подход, основанный на контенте, позволяет использовать другие методы семантического поиска, рассмотренные в главах 5–7. Для простоты мы будем использовать здесь подход, основанный на модели, и отдадим должное LLM для определения смысла запроса. Следующий листинг демонстрирует три различных подхода к получению топовых категорий для запроса на основе векторов эмбединга.

Листинг 9.15. Сравнение методов сопоставления запросов с кластерами

```
import sentence_transformers, heapq

def get_top_labels_centers(query, centers, n=2):
    query_embedding = transformer.encode([query], convert_to_tensor=False)
    similarities = sentence_transformers.util.cos_sim(
        query_embedding, centers)
```

Получает топ-N кластеров на основе косинусного сходства с центроидами кластеров.


```

sim = similarities.tolist()[0]
return [sim.index(i) for i in heapq.nlargest(n, sim)]

def get_query_cluster(query):
    query_embedding = transformer.encode([query], convert_to_tensor=False)
    return algo.predict(query_embedding)

def get_cluster_description(cluster_num):
    return "_".join(top_words(clusters[cluster_num], 5))

query = "microwave"
kmeans_predict = get_query_cluster(query)[0]
print("K-means Predicted Cluster:")
print(f"    {kmeans_predict} ({get_cluster_description(kmeans_predict)})")

closest_sim = get_top_labels_centers(query, centers, 1)[0]
print(f"\nCosine Predicted Cluster:")
print(f"    {closest_sim} ({get_cluster_description(closest_sim)})")

knn_cosine_similarity = get_top_labels_centers(query, centers, 5)
print(f"\nKNN Cosine Predicted Clusters: {knn_cosine_similarity}")
for n in knn_cosine_similarity:
    print(f"    {n} ({get_cluster_description(n)})")

```

Получает кластер на основе прогноза модели KMeans.

Вариант 1: прогнозирует ближайший кластер (KMeans).

Вариант 2: находит наиболее похожий кластер (косинусное сходство).

Вариант 3 (рекомендуется): находит N наиболее похожих кластеров (косинусное сходство).

Вывод:

K-means Predicted Cluster:
44 (Microwave_Cu._Ft._Stainless-Steel_Oven)

Cosine Predicted Cluster:
44 (Microwave_Cu._Ft._Stainless-Steel_Oven)

KNN Cosine Predicted Clusters: [44, 52, 5, 83, 6]
44 (Microwave_Cu._Ft._Stainless-Steel_Oven)
52 (Stainless-Steel_30"_Black_Range_Cooktop)
5 (KitchenAid_Black_White_Stand_Mixer)
83 (Black_Coffeemaker_Maker_Coffee_Stainless-Steel)
6 (Range_30"_Self-Cleaning_Freestanding_Stainless-Steel)

В листинге 9.15 мы видим, что вычисляются три прогноза: ближайший кластер (K-средние), наиболее похожий кластер (косинусное сходство) и N наиболее похожих кластеров (косинусное сходство). Функция `get_top_labels_centers` вычисляет топ- N кластеров на основе косинусного сходства с центроидами кластеров¹. Функция кластеризации `get_query_cluster` вычисляет кластер на основе прогноза K-средних.

Выходные данные этих трех подходов демонстрируют важный момент. Хотя запрос относится к `microwave`, мы знаем, что категории

¹ Центроид кластера – это последовательность, наиболее репрезентативная среди остальных последовательностей, назначенных кластеру. – *Прим. ред.*

были сгенерированы динамически и могут пересекаться между продуктами. В этом примере и метод К-средних, и подходы косинусного сходства выбирают категорию 44 (`Microwave_Cu._Ft._Stainless-Steel_Oven`). Хотя вы, скорее всего, найдете лучшие результаты, полагаясь на косинусное сходство для измерения семантического сходства по сравнению с прогнозом К-средних, категории, возвращаемые из каждого из них, скорее всего, будут тесно связаны. Таким образом, любая персонализация выиграет, если будет применена к *каждой* из соответствующих категорий, а не только к одной. Продукты можно разделить на несколько связанных категорий, а значимые категории можно произвольно разделить на основе количества предметов и нюансов описаний предметов.

Чтобы преодолеть перекрытия между похожими категориями, мы рекомендуем использовать кластеры с предсказаниями косинуса топ- N (Knn , вариант 3) вместо фильтрации в один кластер. В результатах из листинга 9.15 этот смешанный подход возвращает пять связанных категорий: 44 («микроволновки»), 52 («плиты»), 5 («разные приборы»), 83 («приборы для столешниц») и 33 («духовки»). Далее мы будем использовать эти прогнозируемые категории вместе с эмбедингами из предыдущих взаимодействий пользователя для персонализации результатов поиска.

9.4.3. Интеграция персонализации на основе эмбедингов в результаты поиска

Последний шаг в нашем пути персонализации – выполнение персонализированного поиска. Мы могли бы сделать это разными способами:

- выполнить средневзвешенное значение между вектором запроса (эмбединг для `microwave`) и векторами предыдущих взаимодействий пользователя в прогнозируемых кластерах. Это сгенерирует один вектор, представляющий персонализированную версию запроса пользователя, поэтому все результаты будут персонализированными;
- выполнить стандартный поиск, но затем повысить результаты на основе среднего значения эмбедингов из предыдущих взаимодействий пользователя в прогнозируемых кластерах. Это будет гибридная функция ранжирования на основе ключевых слов и векторов, где поиск по ключевым словам будет основным драйвером результатов, но предыдущие взаимодействия пользователя будут использоваться для повышения связанных результатов;
- выполнить одно из вышеперечисленных действий, но затем персонализировать только несколько предметов в результатах поиска вместо всех результатов. Это следует принципу легкого прикосновения¹, не нарушать все результаты поиска пользователя,

¹ Принцип *Light-touch* в программировании относится к алгоритму оптимизации с большим количеством ограничений. Он позволяет достигать хорошей скорости сходимости без частой проверки всех ограничений на каждой итерации. Для этого обучают вероятностное распределение по ограничениям, которое концентрируется

при этом все еще внося новизну, чтобы пользователь мог обнаружить персонализированные предметы, которые он, возможно, не нашел бы в противном случае;

- выполнить стандартный поиск (по ключевому слову или вектору), но затем переранжировать результаты на основе средневзвешенного значения между вектором запроса и векторами для предыдущих взаимодействий пользователя в пределах прогнозируемых кластеров. Это использует исходный поиск для поиска результатов-кандидатов с использованием алгоритма релевантности по умолчанию, но эти результаты переранжируются, чтобы повысить персонализированные предпочтения.

Мы продемонстрируем последний метод, поскольку его легко воспроизвести в любой поисковой системе, так как проход персонализации и реранкинга можно выполнить в качестве последнего шага после исходного поиска. Таким образом, этот метод будет хорошо работать как с традиционными поисковыми системами, так и с векторными базами данных.

В листинге 9.16 показаны две ключевые функции, которые мы будем использовать для генерации нашего вектора персонализации: функция `get_user_embeddings`, которая ищет эмбединги для списка продуктов, а также возвращает кластер, связанный с каждым продуктом, и функция `get_personalization_vector`, которая может объединять эмбединги между запросом и всеми соответствующими векторами взаимодействия пользователя и предмета.

Листинг 9.16. Функции для генерации векторов персонализации

```
def top_clusters_for_embedding(embedding, n=2):
    similarities = sentence_transformers.util.cos_sim(embedding, centers)
    sim = similarities.tolist()[0]
    return [sim.index(i) for i in heapq.nlargest(n, sim)]

def get_user_embeddings(products=[]):
    values = []
    embeddings = get_indexed_product_embeddings()
    for p in products:
        values.append([embeddings[p],
                       top_clusters_for_embedding(embeddings[p], 1)[0]])
    return pandas.DataFrame(data=numpy.array(values), index=products,
                           columns=["embedding", "cluster"])

def get_personalization_vector(query=None,
                               user_items=[]),
```

Возвращает датафрейм с эмбедингом и ограниченным кластером для каждого продукта.

Возвращает вектор, который объединяет (взвешенный средний) эмбединг для запроса с эмбедингами для переданных user_items.

на наиболее нарушенных, и на каждой итерации выбирают ограничения из этого распределенного множества. Такой подход называют Light-touch, потому что на каждой итерации проверяется только несколько ограничений, а решение лишь подталкивается к допустимому множеству. – *Прим. ред.*

```

        query_weight=1,
        user_items_weights=[]):
    query_embedding = transformer.encode(query) if query else None

    if len(user_items) > 0 and len(user_items_weights) == 0:
        user_items_weights = numpy.full(shape=len(user_items),
                                         fill_value=1 / len(user_items))

    embeddings = []
    embedding_weights = []
    for weight in user_items_weights:
        embedding_weights.append(weight)
    for embedding in user_items:
        embeddings.append(embedding)
    if query_embedding.any():
        embedding_weights.append(query_weight)
        embeddings.append(query_embedding)

    return numpy.average(embeddings, weights=numpy.array(embedding_weights),
                        axis=0).astype("double") if len(embeddings) else None

```

По умолчанию вес делится 1:1 (по 50 % каждый) между эмбеддингом запроса и user_items_weight.

Вы можете дополнительно указать query_weight и user_item_weights, чтобы повлиять на то, насколько каждый эмбеддинг влияет на вектор персонализации.

С возможностью объединения эмбеддингов и поиска ограниченного кластера (guardrail cluster) для любого продукта пришло время сгенерировать вектор персонализации для пользователя на основе его входящего запроса и прошлых взаимодействий с продуктами. Мы сгенерируем вектор персонализации с ограничениями, а также один без ограничений, чтобы сравнить результаты непосредственно.

Листинг 9.17 демонстрирует, как генерировать векторы персонализации. В этом случае пользователь ранее взаимодействовал с двумя продуктами: бутылкой для воды Hello Kitty и электрической плитой из нержавеющей стали. Теперь они запускают новый запрос по ключевому слову microwave.

Листинг 9.17. Генерация векторов персонализации из пользовательских запросов

```
product_interests = ["7610465823828", #hello kitty water bottle
                    "36725569478"] #stainless steel electric range
```

```

user_embeddings = get_user_embeddings(product_interests)
query = "microwave"

unfiltered_personalization_vector =
    get_personalization_vector(query=query,
                              user_items=user_embeddings['embedding'].to_numpy())
print("\nPersonalization Vector (No Cluster Guardrails):")
print(format_vector(unfiltered_personalization_vector))

```

Вектор персонализации без ограничений (использует запрос и все прошлые взаимодействия с предметами).

```

query_clusters = get_top_labels_centers(query,
                                       centers, n=5)
print("\nQuery Clusters ('microwave'):\n" + str(query_clusters))

Получает 5 верхних кластеров для запроса, кото-  
рые будут использоваться в качестве ограничений.

Фильтрует только предметы в огра-  
ниченных кластерах запроса.

clustered = user_embeddings.cluster.isin(query_clusters)
products_in_cluster = user_embeddings[clustered]
print("\nProducts Filtered to Query Clusters:\n" + str(products_in_cluster))

filtered_personalization_vector = get_personalization_vector(query=query,
                                                             user_items=filtered['embedding'].to_numpy())
print("\nFiltered Personalization Vector (With Cluster Guardrails):")
print(format_vector(filtered_personalization_vector))

Генерирует вектор персонализации с огра-  
ничениями (использует запрос и только  
предметы, связанные с запросом).

Вывод:

Products Interactions for Personalization:
product      embedding
cluster
7610465823828 [0.06417941, 0.04178553, -0.0017139615, -0.020...    1
36725569478   [0.0055417763, -0.024302201, -0.024139373, -0....    6

Personalization Vector (No Cluster Guardrails):
[0.016, -0.006, -0.02, -0.032, -0.016, 0.008, -0.0, 0.017, 0.011, 0.007 ...]

Query Clusters ('microwave'):
[44, 52, 5, 83, 6]

Products Filtered to Query Clusters:
product      embedding
cluster
36725569478   [0.0055417763, -0.024302201, -0.024139373, -0....    6

Filtered Personalization Vector (With Cluster Guardrails):
[0.002, -0.023, -0.026, -0.037, -0.025, 0.002, -0.009, 0.007, 0.033, -0 ...]

```

Листинг 9.17 выполняет четырехэтапный процесс для создания вектора персонализации для запроса пользователя:

- 1 получить список взаимодействий с продуктом вместе с соответствующими эмбедингами продукта и кластерами;
- 2 найти N (в данном случае 5) наиболее похожих кластеров для запроса;
- 3 отфильтровать список взаимодействий пользователя только до предметов в кластерах запроса;
- 4 сгенерировать вектор персонализации (`filtered_personalization_vector`), объединив запрос и отфильтрованные векторы взаимодействия пользователя с предметом. (Примечание: мы также сгенерировали `unfiltered_personalization_vector`, который не применяет категориальные ограничения, для последующего сравнения бок о бок.)

Окончательный вектор `filtered_personalization_vector` можно использовать напрямую для векторного поиска по эмбедингам, поскольку он представляет эмбединг для запроса, который был представлен в соответствии с интересами пользователя в 768-мерном векторном пространстве эмбединга. В нашем случае мы вместо этого запустим независимый поиск для запроса, а затем используем `filtered_personalization_vector` для реранкинга лучших результатов. Следующий листинг демонстрирует этот процесс поиска и переранжирования.

Листинг 9.18. Использование вектора персонализации для реранкинга результатов

```
def rerank_with_personalization(docs,
                                personalization_vector):
    embeddings = get_indexed_product_embeddings()
    result_embeddings = numpy.array(
        [embeddings[docs[x]]["upc"]]
        for x in range(len(docs))]).astype(float)
    similarities = sentence_transformers.util.cos_sim(
        personalization_vector, result_embeddings).tolist()[0]
    reranked = [similarities.index(i)
                 for i in heapq.nlargest(len(similarities), similarities)]
    reranked, _ = zip(*sorted(enumerate(similarities),
                               key=itemgetter(1), reverse=True))
    return [docs[i] for i in reranked]
```

Переранжирует все результаты поиска на основе косинусного сходства с персонализированным вектором запроса.

```
query = "microwave"
request = product_search_request(query, {"limit": 100})

response = products_collection.search(**request)
docs = response["docs"]
print("No Personalization:")
display_product_search(query, docs[0:4])
```

Отображает исходные результаты поиска (без персонализации).

```
print("Global Personalization (No Category Guardrails):")
reranked_seach_results_no_guardrails = \
    rerank_with_personalization(docs,
                                unfiltered_personalization_vector)
display_product_search(query, reranked_seach_results_no_guardrails[0:4])
```

Персонализированный поиск без ограничений (использует `unfiltered_personalization_vector`).

```
print("Contextual Personalization (with Category Guardrails):")
reranked_seach_results_with_guardrails = \
    rerank_with_personalization(docs,
                                filtered_personalization_vector)
display_product_search(query, reranked_seach_results_with_guardrails[0:4])
```

Персонализированный поиск с ограничениями (использует `filtered_personalization_vector`).

В листинге 9.18 показан весь процесс применения векторов персонализации для реранкинга результатов поиска. Функция `rerank_with_personalization` берет исходные результаты поиска и вектор персонализации, а затем переранжирует результаты поиска на основе

косинусного сходства между вектором персонализации и векторами эмбединга для каждого результата поиска. Мы вызываем реранкинг дважды для целей сравнения: один раз с ограничениями, примененными к вектору персонализации, и один раз без них. Каждый из окончательных наборов ранжированных результатов передается в функцию `display_product_search` для отображения трех наборов результатов, сравниваемых на рис. 9.16: персонaлизированные результаты поиска, персонaлизированные результаты поиска без ограничений и персонaлизированные результаты поиска с ограничениями.

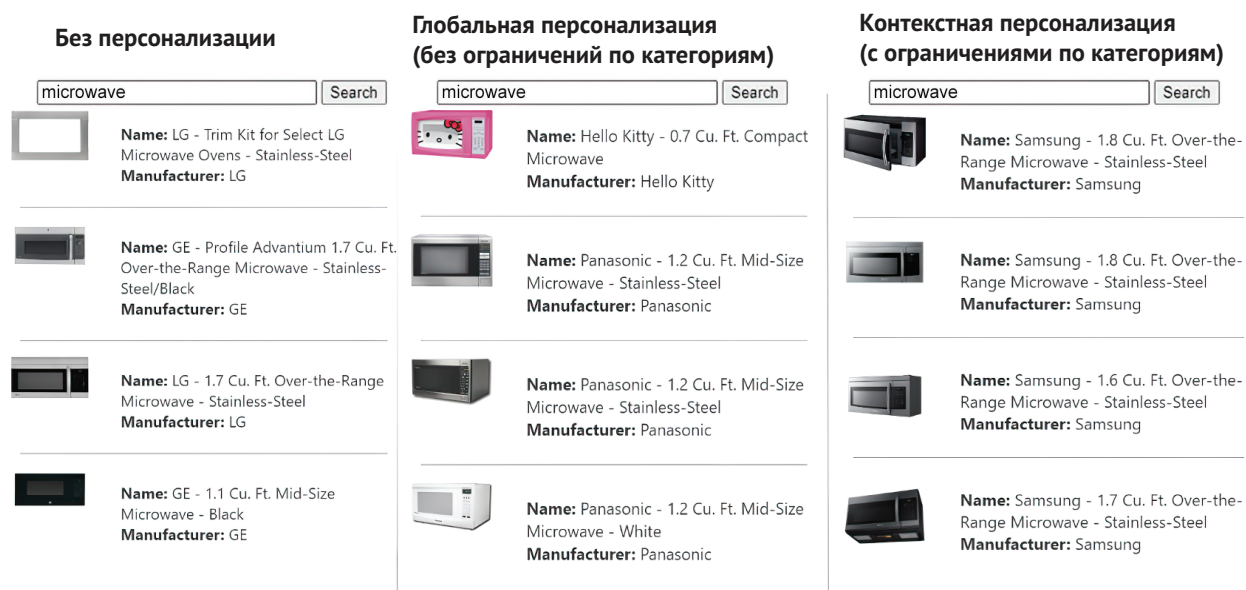


Рис. 9.16. Сравнение персонaлизированных, всегда персонaлизированных (без ограничений) и контекстно персонaлизированных (с ограничениями) результатов поиска

Слева мы видим исходные результаты поиска для `microwave`, включая крышку для микроволновой печи, некоторые микроволновые печи из нержавеющей стали и простую микроволновую печь. В середине мы видим персонaлизированные результаты поиска без категориальных ограничений. Вектор персонaлизации пользователя включает в себя эмбединги для микроволновой печи из нержавеющей стали, а также бутылки для воды Hello Kitty. Как вы можете видеть, микроволновая печь Hello Kitty сразу перешла на верх результатов, хотя пользователь ранее смотрел на холодильник из нержавеющей стали, и его интерес к бутылке для воды вряд ли перейдет в интерес к микроволновой печи Hello Kitty. Справа мы видим персонaлизацию с примененными ограничениями. Мы видим, что все эти результаты теперь для микроволновых печей из нержавеющей стали, что отражает предыдущий интерес пользователя к холодильнику из нержавеющей стали, который был автоматически определен как похожая категория. Теперь вы реализовали сквозной персонaлизированный алгоритм поиска. Персонaлизированный поиск может значительно повысить релевантность, если внедрять его осторожно и с легким прикосновением, но важно не разочаровывать пользователей чрез-

мерной персонализацией. В следующем разделе мы рассмотрим некоторые подводные камни и проблемы с персонализацией, которые вам нужно иметь в виду, чтобы избежать потенциального разочарования пользователей.

9.5. Проблемы с персонализацией результатов поиска

В этой главе мы выделили многие проблемы с персонализацией результатов поиска. Хотя персонализация может быть мощным инструментом для получения более релевантных результатов поиска, важно знать о потенциальных подводных камнях и гарантировать, что персонализация применяется только тогда, когда она, скорее всего, будет полезна пользователю. Мы затронули следующие ключевые проблемы в этой главе:

- проблему холодного запуска – при использовании совместной фильтрации пользователи, которые не взаимодействовали ни с одним предметом, не имеют никакой информации, на которой можно было бы основывать персонализацию. Для таких пользователей важно вернуться к неперсонализированным результатам поиска. Объединение подхода к фильтрации на основе контента (поиск или сопоставление на основе признаков) с совместной фильтрацией может помочь преодолеть проблему холодного старта;
- ограничения важны – применение персонализации через категориальные границы, как правило, плохая идея. В противном случае, когда пользователь переключает контекст, чтобы посмотреть на несвязанные предметы, результаты поиска будут выглядеть странно и будут контрпродуктивными. Просмотр белой бумаги или белых лампочек не означает, что пользователь захочет позже увидеть белые холодильники при поиске бытовой техники. Аналогично, если ему понравился фильм «Терминатор», это не означает, что кто-то захочет купить пистолет или робот-пылесос. При персонализации результатов поиска важно понимать соответствующую область, в которой следует применять изученные предпочтения пользователя. Моделирование связанных категорий для предметов и запросов и ограничение персонализации только использованием предметов, связанных с запросом, – хороший способ избежать этих проблем;
- излишняя персонализация раздражает – когда кто-то вводит поисковый запрос, он ожидает, что поисковая система вернет наиболее релевантные результаты для его конкретного запроса. Хотя применение персонализации может быть очень полезным в определенных случаях использования (например, персонализация локации для ресторана), оно также может быть очень раздражающим, если объем персонализации мешает пользователю контролировать процесс поиска. В качестве крайнего случая представьте, что каждый запрос был бы усилен признаками из

каждого предыдущего запроса или взаимодействия с предметом; процесс поиска быстро бы превратился в беспорядок, который не позволил бы пользователю найти то, что он ищет. Рассмотрите возможность персонализации только нескольких верхних результатов вместо всего набора результатов поиска, чтобы в случае ошибки персонализации неперсонализированные результаты оставались доступными. Также рассмотрите возможность предоставления пользователям возможности отключить персонализацию, если она им не нравится;

- циклы обратной связи имеют решающее значение – интересы пользователей со временем меняются. Независимо от того, показываете ли вы рекомендации или создаете профили персонализации для поиска, пользователи должны иметь возможность предоставлять обратную связь системе, чтобы помочь ей учиться и адаптироваться к их меняющимся интересам. Это можно сделать, разрешив пользователям предоставлять явную обратную связь (например, «палец вверх» или «палец вниз») на рекомендации или просто продолжая собирать неявную обратную связь от поведенческих сигналов (клики, покупки и т. д.) и используя новые взаимодействия для обновления профиля персонализации. В любом случае важно дать пользователям возможность предоставлять обратную связь системе, чтобы она могла учиться и адаптироваться к их меняющимся интересам со временем;
- конфиденциальность может вызывать беспокойство – поскольку персонализация основана на предыдущих моделях взаимодействия пользователя, показ персонализированных рекомендаций и результатов поиска означает сбор и раскрытие прошлого поведения пользователя. Представьте себе службу потокового вещания фильмов, предлагающую жестокие или взрослые фильмы, книжный магазин, предлагающий любовные романы или книги по самосовершенствованию, или продуктовый магазин, продвигающий нездоровую пищу и алкоголь. Это может смущать и деморализовать пользователя, подрывая достоверность и уверенность в сервисе. Важно быть прозрачным в отношении того, какие сигналы собираются и как они используются. Также важно предоставить пользователям возможность отказаться от персонализации, если они обеспокоены своей конфиденциальностью;
- применяйте персонализацию с легким прикосновением – большинство поисковых систем не персонализируют результаты поиска, предоставляя пользователю полный контроль над выражением своих интересов через текущий запрос. Отклонение от этой парадигмы может быть полезным во многих случаях, но важно гарантировать, что персонализация применяется только тогда, когда она, вероятно, будет полезна пользователю. Одна из стратегий обеспечения легкого прикосновения – применять персонализацию только к нескольким верхним результатам. Обычно лучше проявить осторожность с персонализацией и применять

ее очень консервативно. Большинство пользователей будут меньше разочарованы отсутствием персонализации, чем поисковой системой, которая слишком явно старается читать их мысли и при этом ошибается.

Из всех методов поиска на основе ИИ персонализация является как одним из самых недостаточно используемых способов лучшего понимания намерений пользователя, так и одним из самых сложных. Хотя рекомендательные системы распространены, спектр персонализации между поиском и рекомендациями более тонок и менее изучен. Пока персонализированный поиск реализуется с осторожностью, он может быть мощным инструментом для получения более релевантных результатов поиска и экономии времени пользователя на поиск предметов, которые лучше всего соответствуют его конкретным интересам.

Резюме

- Персонализированный поиск находится в середине спектра персонализации между поиском по ключевым словам (управляемым явным вводом пользователя) и совместными рекомендациями (управляемыми неявным вводом, полученным из поведения пользователя).
- Совместные рекомендации можно полностью получить из шаблонов взаимодействия пользователя с документами, но они страдают от проблемы холодного запуска. Объединение совместной фильтрации с признаками на основе контента может преодолеть проблему холодного запуска и обеспечить возможность более гибкого персонализированного поиска.
- Представление документов и пользователей в виде векторов эмбединга позволяет создавать динамические профили персонализации, которые можно использовать для получения более персонализированных результатов поиска.
- Кластеризация продуктов по их векторам эмбединга может использоваться для создания динамических категорий, которые будут служить защитными барьерами для персонализированного поиска, гарантируя, что пользователи не будут видеть результаты, слишком далекие от их интересов.
- Включение циклов обратной связи для обучения на основе взаимодействия с пользователем важно, пока сохраняется конфиденциальность пользователя и оно применяется с легким прикосновением, чтобы избежать чрезмерной персонализации.
- Персонализированный поиск может обеспечить более релевантные результаты поиска, но важно сбалансировать преимущества персонализации с потенциальным разочарованием пользователя, если персонализация слишком агрессивна. Достижение правильного баланса может значительно улучшить понимание поисковой системой намерений пользователя.

10

Обучение ранжированию для обобщаемой релевантности поиска¹

В этой главе рассматривается:

- введение в машинное обучение ранжированию (LTR);
- отличие LTR от других методов машинного обучения;
- обучение и запуск классификатора ранжирования;
- разработка признаков, списки суждений и интеграция моделей машинного обучения ранжирования в поисковую систему;
- проверка модели LTR с использованием разделения обучения и теста;
- компромиссы производительности для моделей ранжирования на основе LTR.

Представьте, что сегодня случайный вторник. Вы просматриваете свои журналы поиска, и поисковые запросы варьируются от запроса обеспокоенного любителя бега о `polar m430 running watch charger` до

¹ Обобщаемая релевантность поиска, англ. *generalizable search relevance*, – это возможность создавать общие системы поиска с помощью машинного обучения. Они позволяют эффективно обрабатывать сложные или двусмысленные запросы и персонализировать результаты поиска. – Прим. ред.

weird bump on nose - cancer? и william shatner first film любопытного киномана. Даже если это могут быть единичные запросы, вы знаете, что каждый пользователь ожидает не меньше чем потрясающих результатов поиска.

Вы ощущаете безнадежность. Вы знаете, что многие строки запросов сами по себе удручающе редки. У вас очень мало данных о кликах, чтобы знать, что релевантно для этих поисков. Каждый день становится все сложнее: тенденции, варианты использования, продукты, пользовательские интерфейсы и даже пользовательская терминология развиваются. Как кто-то может надеяться создать поиск, который восхищает, когда пользователи, кажется, постоянно удивляют нас новыми вариантами запросов?

Не отчаивайтесь, есть надежда! В этой главе мы познакомимся с обобщаемыми моделями релевантности. Эти модели изучают базовые закономерности, которые управляют ранжированием релевантности. Вместо того чтобы запоминать, что статья под названием «Прыщи: шишки на носу» является ответом на запрос weird bump on nose - cancer? (Странная шишка на носу – это рак?), мы наблюдаем базовую закономерность – сильное соответствие заголовка соответствует высокой релевантности. Если мы сможем изучить эти закономерности и закодировать их в модель, мы сможем выдавать релевантные результаты даже для поисковых запросов, которых мы никогда не видели.

В этой главе рассматривается обучение ранжированию (Learning to rank, LTR): метод, который использует машинное обучение для создания обобщаемых моделей ранжирования релевантности. Мы подготовим, обучим и выполним поиск с помощью моделей LTR, используя поисковую систему.

10.1. Что такое LTR?

Давайте рассмотрим, что делает LTR. Мы увидим, как LTR создает обобщаемые модели ранжирования, находя закономерности, которые предсказывают релевантность. Затем мы рассмотрим более гаяк и болтов построения модели.

10.1.1. Выход за рамки ручной настройки релевантности

Вспомним ручную настройку релевантности из главы 3. Мы наблюдаем факторы, которые соответствуют релевантным результатам, и математически объединяем эти факторы в функцию ранжирования. Функция ранжирования возвращает оценку релевантности, которая упорядочивает результаты как можно ближе к нашему идеальному ранжированию.

Например, рассмотрим поисковую систему фильмов с документами, подобными тем, что перечислены в следующем списке. Этот документ взят из корпуса TheMovieDB (tmdb) (<http://themoviedb.org>), который мы будем использовать в этой главе. Если вы хотите следовать коду для этой главы, используйте первый блокнот этой главы для индексации набора данных tmdb.

Листинг 10.1. Документ для фильма «Социальная сеть»

```
{
  "title": ["The Social Network"],
  "overview": [
    "On a fall night in 2003, Harvard undergrad and computer  

    ➔programming genius Mark Zuckerberg sits down at his computer and  

    ➔heatedly begins working on a new idea. In a fury of blogging and  

    ➔programming, what begins in his dorm room as a small site among  

    ➔friends soon becomes a global social network and a revolution in  

    ➔communication. A mere six years and 500 million friends later,  

    ➔Mark Zuckerberg is the youngest billionaire in history... but for  

    ➔this entrepreneur, success leads to both personal and legal  

    ➔complications."],
  "tagline": ["You don't get to 500 million friends without making a few  

    ➔enemies."],
  "release_year": 2010}

```

Благодаря бесконечным итерациям и настройкам мы можем прийти к обобщенной функции ранжирования фильмов, которая выглядит примерно так, как в следующем листинге.

Листинг 10.2. Обобщенная функция ранжирования с использованием ручных бустов

```
keywords = "some example keywords"
{"query": f"title:({keywords})^10 overview:({keywords})^20  

➔{!func}release_year^0.01"}

```

Ручная оптимизация весов признаков общих функций ранжирования, подобных этой, для работы со многими запросами может потребовать значительных усилий, но такие оптимизации идеально подходят для машинного обучения.

Именно здесь в игру вступает LTR – он берет наши предложенные факторы релевантности и находит оптимальную функцию ранжирования. LTR принимает несколько форм, от простого набора линейных весов (например, бустов здесь) до сложной модели глубокого обучения.

Чтобы изучить основы, в этой главе мы построим простую модель LTR. Мы найдем оптимальные веса для title, overview и release_year в оценочной функции, подобной той, что в листинге 10.2. С помощью этой относительно простой задачи мы увидим полный жизненный цикл разработки решения LTR.

10.1.2. Реализация LTR в реальном мире

Поскольку мы продолжаем определять LTR на высоком уровне, давайте быстро проясним, где LTR вписывается в общую картину поисковой системы. Затем мы можем рассмотреть типы данных, которые нам понадобятся для построения модели LTR.

Мы сосредоточимся на построении LTR для продуктивных поисковых систем, которые могут сильно отличаться от исследовательского

контекста. Нам нужны не только релевантные результаты, но и результаты, возвращаемые достаточно быстро, с использованием основных, хорошо понятных методов поиска.

Концептуально вызов LTR обычно включает три высокоуровневых шага:

- 1 обучение модели LTR;
- 2 запуск модели в производстве;
- 3 использование модели для ранжирования (или реранкинга) результатов поиска.

Большинство современных поисковых систем поддерживают запуск моделей ранжирования непосредственно в поисковой системе, что позволяет эффективно вызывать модель LTR «там, где находятся данные». Обычно модели LTR значительно медленнее при ранжировании, чем базовые функции ранжирования на основе ключевых слов, такие как BM25, поэтому модели LTR часто вызываются только для ранжирования (или реранкинга) последующего прохода на подмножестве верхних результатов поиска, ранжированных первоначальной, более быстрой функцией ранжирования. Вставка модели LTR в движок (если поддерживается) устраняет необходимость возвращать сотни или тысячи документов и их метаданные из поисковой системы во внешнюю службу моделей для реранкинга, что может быть медленным и неэффективным по сравнению с выполнением работы в движке и в масштабе.

По этой причине наша библиотека `ltr` в этой главе реализует подключаемую поддержку для развертывания и вызова собственных возможностей интеграции модели LTR каждой поддерживаемой поисковой системы или векторной базы данных, если они доступны. Код в каждом листинге будет работать с любым поддерживаемым движком (см. приложение В, чтобы изменить его), но вывод листинга, который вы увидите в этой главе, будет отражать реализацию LTR Solr, поскольку Solr настроен по умолчанию. Если вы измените движок, то увидите вывод выбранного вами движка при запуске блокнотов Jupyter.

Solr был одной из первых крупных поисковых систем с открытым исходным кодом, изначально поддерживающих обслуживание модели LTR, а позже эти возможности были перенесены в разработанный сообществом плагин Elasticsearch LTR (<https://github.com/o19s/elasticsearch-learning-to-rank>), а затем разветвлены на плагин OpenSearch LTR (<https://github.com/opensearch-project/opensearch-learning-to-rank-base>). Таким образом, плагины Elasticsearch и OpenSearch LTR реализуют почти те же понятия, что и Solr. Vespa реализует поэтапное ранжирование (или реранкинг) и возможность вызывать модели во время каждой фазы, а Weaviate также реализует различные возможности реранкинга. Другие движки, поддерживающие собственный LTR, будут следовать аналогичным шаблонам.

Рисунок 10.1 описывает рабочий процесс для разработки практического решения LTR.

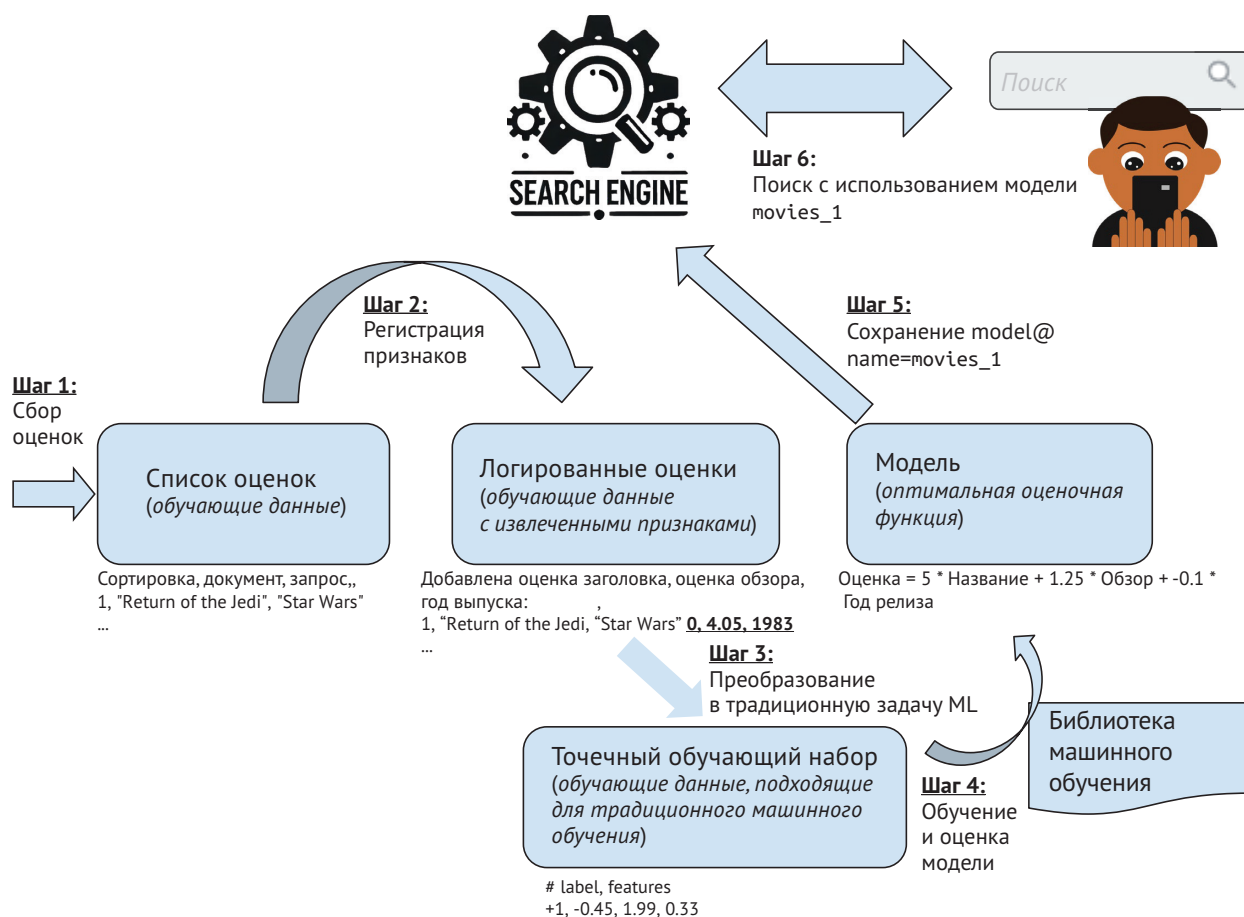


Рис. 10.1. Системы LTR преобразуют наши обучающие данные (списки суждений) в модели, которые обобщают рейтинг релевантности. Этот тип системы позволяет нам находить базовые закономерности в наших обучающих данных

Вы можете заметить сходство между LTR и традиционными рабочими процессами классификации или регрессии на основе машинного обучения. Но исключения – это то, что делает их интересными. Таблица 10.1 отображает определения между традиционными целями машинного обучения и LTR.

Эта глава следует шагам на рис. 10.1 для обучения модели LTR.

- 1 *Сбор суждений* – мы получаем суждения (judgements) из кликов или других источников. Мы подробно рассмотрим этот шаг в главе 11.
- 2 *Регистрация признаков* – чтобы обучить модель, мы должны объединить суждения с признаками, чтобы увидеть общую закономерность. Этот шаг требует от нас попросить поисковую систему сохранить и вычислить запросы, представляющие признаки.
- 3 *Преобразование в традиционную задачу машинного обучения* – вы увидите, что LTR на самом деле заключается в переводе задачи ранжирования во что-то, что больше похоже на столбец «традиционное машинное обучение» в табл. 10.1.
- 4 *Обучение и оценка модели* – здесь мы создаем нашу модель и подтверждаем, что она действительно обобщаема и, таким образом, будет хорошо работать для запросов, которые она не видела.
- 5 *Сохранение модели* – мы загружаем модель в нашу поисковую инфра-

структуру, сообщаем поисковой системе, какие признаки использовать в качестве входных данных, и предоставляем ее пользователям для использования в своих поисках.

- 6 *Поиск с использованием модели* – наконец-то мы можем выполнять поиск с использованием модели!

Таблица 10.1. Традиционное машинное обучение против LTR

Понятие	Традиционное машинное обучение	LTR
Данные для обучения	Набор исторических или «истинных» примеров, которые модель должна попытаться предсказать, например цены акций за прошедшие дни, например «Apple» стоила 125 долл. 6 июня 2021 года	Список суждений: суждение просто маркирует документ как релевантный или нерелевантный для запроса. На рис. 10.2 <i>Return of the Jedi</i> помечен как релевантный (оценка 1) для запроса <code>star wars</code>
Признак	Данные, которые мы можем использовать для прогнозирования обучающих данных, например у Apple было 147 000 сотрудников и доход в размере 90 млрд долл.	Данные используются так, чтобы релевантные результаты, ранжированные выше нерелевантных и в идеале значений, которые поисковая система может вычислить быстро. Наши функции – это поисковые запросы типа <code>title:({keywords})</code> из листинга 10.2
Модель	Алгоритм, который использует признаки в качестве входных данных для прогнозирования. Учитывая, что на 6 июля 2021 года в Apple работает 157 000 человек, а выручка составляет 95 млрд долл., модель может предсказать цену акций на эту дату в размере 135 долл.	Объединяет ранжирующие функции (поисковые запросы) вместе, чтобы назначить оценку релевантности каждому потенциальному результату поиска. Результаты сортируются по убыванию оценки с надеждой, что более релевантные результаты будут размещены первыми

В оставшейся части главы мы подробно рассмотрим каждый из этих шагов, чтобы построить нашу первую реализацию LTR. Давайте начнем!

10.2. Шаг 1: список суждений, начиная с обучающих данных

Вы уже видели, что такое LTR на высоком уровне, поэтому давайте перейдем к сути. Перед реализацией LTR мы должны сначала узнать о данных, используемых для обучения модели LTR: списке суждений.

Список суждений – это список меток релевантности или *оценок* (grades), каждая из которых указывает на релевантность документа запросу. Оценки могут иметь различные формы. Пока мы будем придерживаться простых *бинарных суждений* – 0 указывает на нерелевантный документ, а 1 – на релевантный.

Используя класс Judgment, предоставленный в коде этой книги, мы пометим «The Social Network» как релевантный для запроса `social network`, создав Judgment:

```
from ltr.judgments import Judgment
Judgment(grade=1, keywords="social network", doc_id=37799)
```

Интереснее посмотреть несколько запросов. В листинге 10.3 у нас есть `social network` и `star wars` как два разных запроса с фильмами, оцененными как релевантные или нерелевантные.

Листинг 10.3. Маркировка суждений о фильмах как релевантных или нерелевантных

```
sample_judgments = [
    # for 'social network' query
    Judgment(1, "social network", 37799), # The Social Network
    Judgment(0, "social network", 267752), # #chicagoGirl
    Judgment(0, "social network", 38408), # Life As We Know It
    Judgment(0, "social network", 28303), # The Cheyenne Social Club

    # for 'star wars' query
    Judgment(1, "star wars", 11), # Star Wars
    Judgment(1, "star wars", 1892), # Return of the Jedi
    Judgment(0, "star wars", 54138), # Star Trek Into Darkness
    Judgment(0, "star wars", 85783), # The Star
    Judgment(0, "star wars", 325553) # Battlestar Galactica
]
```

Вы можете видеть, что мы пометили «Star Trek Into Darkness» и «Battlestar Galactica» как нерелевантные для запроса `star wars`, но «Return of the Jedi» как релевантные.

Надеюсь, вы спрашиваете себя: «Откуда взялись эти оценки?» Маркированы вручную экспертами по кино? На основе кликов пользователей? Хорошие вопросы! Создание хорошего обучающего набора на основе взаимодействия пользователей с результатами поиска имеет решающее значение для того, чтобы LTR работал хорошо. Чтобы получить данные для обучения в большом объеме, мы обычно получаем эти метки из трафика кликов с помощью типа алгоритма, известного как *модель кликов*¹. Поскольку этот шаг является настолько основополагающим, мы посвятим всю главу 11 более глубокому погружению в тему. Однако в этой главе мы начнем с оценочных суждений, маркированных вручную, чтобы изначально сосредоточиться на механике LTR.

Каждое суждение также имеет вектор признаков, который можно использовать для обучения модели. Первый признак в векторе признаков может соответствовать оценке `title BM25`, второй – оценке `overview BM25` и т. д. Мы еще не заполняли векторы признаков, поэтому, если вы проверите `sample_judgments[0].features`, он в настоящее время пуст (`[]`).

Давайте воспользуемся поисковой системой, чтобы собрать некоторые признаки.

¹ Модель кликов – это предсказательная вероятностная модель поведения пользователя в поиске, которая стремится предсказать будущие шаблоны поведения на основе анализа исторических данных. – *Прим. ред.*

10.3. Шаг 2: логирование признаков и инжиниринг

Проектирование признаков требует выявления закономерностей между признаками документа и релевантностью. Например, мы можем выдвинуть гипотезу, что релевантные результаты в наших суждениях соответствуют сильным совпадениям заголовков. В этом случае совпадение заголовков будет признаком, который нам нужно будет определить. В этом разделе вы увидите, что такое признаки (например, совпадение заголовков) и как использовать современную поисковую систему для проектирования и извлечения этих признаков из корпуса.

Для целей LTR *признак* – это некоторый числовой атрибут документа, запроса или отношения запрос–документ. Признаки – это математические строительные блоки, которые мы используем для построения функции ранжирования. Вы уже видели ручную функцию ранжирования с функциями в листинге 10.2: оценка ключевого слова в поле title является одной из таких функций, как и оценки ключевых слов release_year и overview:

```
"query": f"title:({keywords})^10 overview:({keywords})^20
➡{!func}release_year^0.01"
```

Конечно, признаки, которые вы в конечном итоге используете, могут быть более сложными или домен-специфичными, например расстояние от места жительства до работы при поиске работы или некоторая связь графа знаний между запросом и документом. Все, что вы можете вычислить относительно быстро, когда пользователь выполняет поиск, может быть разумным признаком.

Логирование признаков берет список суждений и вычисляет признаки для каждой помеченной пары запрос–документ. Если бы мы вычислили значения каждого компонента листинга 10.2 для запроса социальной сети, мы бы получили что-то вроде табл. 10.2.

Таблица 10.2. Признаки, логированные для ключевых слов social network для релевантных (grade=1) и нерелевантных (grade=0) документов

Сортировка	Фильм	title:({keywords})	overview:({keywords})	{!func}release_year
1	Social Network	8.243603	3.8143613	2010.0
0	#chicagoGirl	0.0	6.0172443	2013.0
0	Life As We Know It	0.0	4.353118	2010.0
0	The Cheyenne	3.4286604	3.1086721	1970.0
	Social Club			

Алгоритм машинного обучения может изучить значения признаков из табл. 10.2 и прийти к хорошей функции ранжирования. Из данных

только табл. 10.2 кажется, что такой алгоритм может создать функцию ранжирования с более высоким весом для заголовка признака и более низкими весами для других признаков.

10.3.1. Хранение признаков в современном поисковом движке

Современные поисковые движки, которые поддерживают LTR, помогают нам хранить, управлять и извлекать признаки. Такие поисковые системы, как Solr, Elasticsearch и OpenSearch, отслеживают признаки в *хранилище признаков* – списке именованных признаков. Крайне важно, чтобы мы регистрировали признаки для обучения способом, соответствующим тому, как поисковая система будет выполнять модель.

Как показано в листинге 10.4, мы генерируем и загружаем признаки в поисковую систему. Мы используем общую абстракцию хранилища признаков в кодовой базе книги, что позволяет нам генерировать различные признаки на основе поиска и загружать их как *набор признаков* в хранилище признаков поддерживаемой поисковой системы. Здесь мы создаем три признака: оценку релевантности поля заголовка `title_bm25`, оценку релевантности поля обзора `overview_bm25` и значение поля `release_year`. BM25 здесь соответствует оценке на основе BM25, определенной в главе 3, и будет нашим методом по умолчанию для оценки совпадений терминов в текстовых полях.

Листинг 10.4. Создание трех признаков для LTR

```
feature_set = [
    ltr.generate_query_feature (feature_name="title_bm25",
                               field_name="title"),
    ltr.generate_query_feature (feature_name="overview_bm25",
                               field_name="overview"),
    ltr.generate_field_value_feature (feature_name="release_year",
                                     field_name="release_year")]

ltr.upload_features(features=feature_set, model_name="movie_model")
display(feature_set)
```

Определение набора движков-специфичных признаков (для engine=solr):

```
[{"name": "title_bm25",
  "store": "movies",
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "title:${keywords}"}}],
{"name": "overview_bm25",
  "store": "movies",
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "overview:${keywords}"}}],
{"name": "release_year",
  "store": "movies",
  "class": "org.apache.solr.ltr.feature.SolrFeature",
  "params": {"q": "release_year:${keywords}"}}]
```

Название признака.

Хранилище признаков, в котором будет сохранен признак.

Параметризованный признак, берущий ключевые слова (например, «star wars») и ищущий поле заголовка.

Еще один признак, который выполняет поиск по полю обзора.

Только признак документа, release_year фильма.


```
"class": "org.apache.solr.ltr.feature.SolrFeature",
"params": {"q": "{!func}release_year"}]]
```

Параметры – это те же параметры для запроса Solr, что позволяет использовать всю мощь обширного Query DSL Solr для создания признаков.

Вывод листинга 10.4 показывает набор признаков, который загружается в поисковую систему, в данном случае набор признаков Solr. Этот вывод, очевидно, будет выглядеть по-разному в зависимости от того, какую реализацию поисковой системы вы настроите (как обсуждается в приложении В). Первые два признака параметризованы: каждый из них берет ключевые слова поиска (social network, star wars) и выполняет поиск по соответствующему полю. Последний признак – это признак значения поля, использующий год выпуска фильма, что поднимет рейтинг более свежих фильмов выше.

10.3.2. Логирование признаков из корпуса нашего поискового движка

С признаками, загруженными в поисковую систему, нашей следующей задачей будет логирование признаков для каждой строки в нашем списке суждений. После того как мы разберемся с этой последней частью «сан-техники», мы обучим модель, которая может наблюдать связи между каждым релевантным и нерелевантным документом для каждого запроса.

Для каждого уникального запроса в нашем списке суждений нам нужно извлечь признаки для оцененных документов запроса. Для запроса social network в списке образцов суждений из листинга 10.3 у нас есть один релевантный документ (37799) и три нерелевантных документа (267752, 38408 и 28303).

В следующем листинге показан пример логирования признаков для запроса social network.

Листинг 10.5. Логирование значений признаков для результатов social network

```
ids = ["37799", "267752", "38408", "28303"]
options = {"keywords": "social network"}
ltr.get_logged_features("movie_model", ids,
                      options=options,
                      fields=["id", "title"])
display(response)
```

Релевантные и нерелевантные документы для запроса «social network».

Запрашивает у поисковой системы значения признаков, содержащиеся в хранилище признаков фильмов.

Движок-специфичный запрос на поиск (для engine=solr):

```
{"query": "{!terms f=id}37799,267752,38408,28303",
"fields": ["id", "title",
' [features store=movies efi.keywords="social network"] ' ]}]
```

Пример синтаксиса запроса Solr для извлечения значений признаков из каждого возвращенного документа.

Документы с логированными признаками:

```
[{"id": "37799",
  "title": "The Social Network",
  "[features]": {"title_bm25": 8.243603,
                  "overview_bm25": 3.8143613,
                  "release_year": 2010.0}},
 {"id": "267752",
  "title": "#chicagoGirl",
  "[features]": {"title_bm25": 0.0,
                  "overview_bm25": 6.0172443,
                  "release_year": 2013.0}},
 {"id": "38408",
  "title": "Life As We Know It",
  "[features]": {"title_bm25": 0.0,
                  "overview_bm25": 4.353118,
                  "release_year": 2010.0}},
 {"id": "28303",
  "title": "The Cheyenne Social Club",
  "[features]": {"title_bm25": 3.4286604,
                  "overview_bm25": 3.1086721,
                  "release_year": 1970.0}}]
```

Каждое значение признака, логированное для этого фильма для запроса «social network».

Обратите внимание, что поисковый запрос (в данном случае для Solr) в листинге 10.5 имеет возвращаемое поле, содержащее квадратные скобки. Этот синтаксис сообщает Solr о необходимости возвращать дополнительное поле для каждого документа, содержащего данные признаков, определенные в хранилище признаков (в данном случае хранилище признаков `movies`). Параметр `efi` обозначает *внешнюю информацию о признаках* и используется здесь для передачи запроса ключевого слова (`social network`) и любой дополнительной информации о времени запроса, необходимой для вычисления каждого признака. Ответ содержит четыре запрошенных документа с соответствующими им признаками. Эти параметры будут разными для каждой поисковой системы, но понятия будут похожими.

С помощью некоторого обыденного преобразования данных Python мы можем заполнить признаки для запроса `social network` в нашем обучающем наборе из этого ответа. В листинге 10.6 мы применяем данные о признаках к суждениям для запроса `social network`:

Листинг 10.6. Суждения с логированными признаками для запроса `social network`

Суждение для фильма «Социальная сеть» относительно запроса «social network», включая логированные значения признаков.

```
Judgment(grade=1, keywords="social network", doc_id=37799, qid=1,
          features=[8.243603, 3.8143613, 2010.0], weight=1),
Judgment(0, "social network", 267752, 1, [0.0, 6.0172443, 2013.0], 1),
```

```
Judgment(0, "social network", 38408, 1, [0.0, 4.353118, 2010.0], 1),  
Judgment(0, "social network", 28303, 1, [3.4286604, 3.1086721, 1970.0], 1)]
```

**Нерелевантный документ для запроса «social network»
(обратите внимание на низкое значение первого признака, оценка title_bm25 0.0).**

В листинге 10.6, как и следовало ожидать, первое значение признака соответствует первому признаку в нашем хранилище признаков (title_bm25), второе значение – второму признаку в нашем хранилище признаков (overview_bm25) и т. д. Давайте повторим процесс логирования признаков для суждений по запросу star wars.

Листинг 10.7. Логированные суждения по запросу star wars

```
[Judgment(1, "star wars", 11, 2, [6.7963624, 0.0, 1977.0], 1),  
Judgment(1, "star wars", 1892, 2, [0.0, 1.9681965, 1983.0], 1),  
Judgment(0, "star wars", 54138, 2, [2.444128, 0.0, 2013.0], 1),  
Judgment(0, "star wars", 85783, 2, [3.1871135, 0.0, 1952.0], 1),  
Judgment(0, "star wars", 325553, 2, [0.0, 0.0, 2003.0], 1)]
```

С возможностью генерировать логированные суждения давайте расширим список суждений примерно до сотни запросов фильмов, каждый из которых будет содержать около 40 фильмов, оцененных как релевантные или нерелевантные. Код для загрузки и логирования признаков для этого большего обучающего набора, по сути, повторяет запрос поисковой системы, показанный в листинге 10.5. Конечный результат логирования признаков выглядит так же, как листинг 10.7, но создан из гораздо большего списка суждений.

Далее мы рассмотрим задачу ранжирования как задачу машинного обучения.

10.4. Шаг 3: преобразование LTR в традиционную задачу машинного обучения

В этом разделе мы рассмотрим ранжирование как задачу машинного обучения. Это поможет нам понять, как применять известные традиционные понятия машинного обучения к нашей задаче LTR.

Задача LTR – просмотреть множество релевантных и нерелевантных обучающих примеров для запроса, а затем построить модель для перемещения более релевантных документов вверх (и наоборот, опустить менее релевантные документы вниз). Каждый обучающий пример сам по себе не имеет большой ценности; важно то, как он упорядочен вместе с другими в запросе. Рисунок 10.2 показывает эту задачу с двумя запросами. Цель состоит в том, чтобы найти оценочную функцию, которая может использовать признаки для правильного упорядочивания результатов.

Ранжирование не является прямым прогнозированием (ранжирование оптимизирует порядок группировки примеров запроса)

qid	grade	title_bm25	overview_bm25
1	0	0.15	0.00
1	0	0.00	0.87
1	1	11.15	9.04
2	0	0.98	3.5
2	1	8.75	5.67
2	0	0.95	4.34



Задача:

Сортировка, поставить релевантное над нерелевантным

```
Score = function(title_bm25,
                  overview_bm25,
                  release_year,
                  ...)
```

Рис. 10.2. LTR – это размещение набора результатов каждого запроса в идеальном порядке, а не прогнозирование индивидуальных оценок релевантности. Это означает, что нам нужно рассматривать каждый запрос как отдельный случай

Сравните LTR с более традиционной задачей точечного машинного обучения: задачей, подобной прогнозированию цены акций компании, как упоминалось в табл. 10.2 ранее. Точечное машинное обучение означает, что мы можем оценить точность модели для каждого примера в отдельности, прогнозируя его абсолютное значение в отличие от его относительного значения по сравнению с другими примерами. Мы знаем, просто глядя на одну компанию, насколько хорошо мы предсказали цену акций этой компании. Сравните рис. 10.3, показывающий точечную задачу, с рис. 10.2. Обратите внимание на рис. 10.3, что обученная функция пытается напрямую предсказать цену акций, тогда как с LTR вывод функции имеет смысл только для упорядочивания предметов относительно их аналогов для запроса.

Традиционное машинное обучение с использованием негруппированного точечного прогнозирования

Stock price	Number of employees	Revenue
\$21.05	1248	\$1.65B
\$915.00	1295	\$590M
\$10.05	98194	\$200M
\$89.58	258	\$23B
\$27.98	45	\$512M
\$34.89	12	\$812M

Задача:

дать правильный точечный прогноз

```
Stock Price = function(num_employees,
                       revenue,
                       ...)
```

Рис. 10.3. Точечное машинное обучение пытается оптимизировать отдельные точечные прогнозы (например, цены акций или температуры). Релевантность поиска – это другая задача, чем точечное машинное обучение. Вместо этого нам нужно оптимизировать ранжирование примеров, сгруппированных по поисковому запросу

LTR преследует совершенно иную цель (ранжирование множества результатов), чем точечное машинное обучение (предсказание специфических значений результатов). Большинство методов LTR используют умную алхимию для преобразования этой задачи «ранжирования пар» в задачу классификации для каждого документа, которая учится предсказывать, какие признаки и веса признаков лучше всего отделяют «релевантные» документы от «нерелевантных». Это преобразование является ключом к построению обобщаемой модели LTR, которая может работать с конкретными документами, а не только с парами документов. Мы рассмотрим метод одной модели для преобразования задачи ранжирования в следующем разделе, исследуя популярную модель LTR под названием SVMrank.

10.4.1. SVMrank: преобразование ранжирования в бинарную классификацию

В основе LTR лежит модель: фактический алгоритм, который изучает связь между релевантностью или нерелевантностью и такими признаками, как `title_bm25`, `overview_bm25` и т. д. В этом разделе мы рассмотрим одну такую модель, SVMrank, для начала объяснив, что означает SVM, а затем как ее можно использовать для построения большой, обобщаемой модели LTR.

SVMrank преобразует релевантность в задачу бинарной классификации. *Бинарная классификация* просто означает классификацию предметов как одного из двух классов (например, «релевантный» против «нерелевантного», «взрослый» против «ребенка», «собака» против «кошки») с использованием доступных признаков.

SVM, или машина опорных векторов, – это один из методов выполнения бинарной классификации. Мы не будем углубляться в SVM, так как вам не нужно быть экспертом в машинном обучении, чтобы следить за обсуждением. Тем не менее, если вы хотите получить более глубокий обзор SVM, вы можете прочитать книгу, например «Грокаем машинное обучение» («Grokking Machine Learning») Луиса Серрано (Manning, 2021)¹.

Интуитивно понятно, что SVM находит лучшую, наиболее обобщаемую гиперплоскость для рисования между двумя классами. *Гиперплоскость* – это граница, которая разделяет векторное пространство на две части. Одномерная точка может быть гиперплоскостью, разделяющей одномерную линию на две части, так же как линия может быть гиперплоскостью, разделяющей двумерную плоскость на две части. Плоскость обычно является трехмерной границей, разделяющей четырехмерное пространство. Все они, а также границы размерностью более трех, в общем называются гиперплоскостями.

¹ Гроккинг в машинном обучении (отложенное обобщение) – это переход к обобщению, который происходит через много итераций обучения после порога интерполяции в отличие от обычного процесса, когда обобщение происходит медленно и прогрессивно после достижения порога интерполяции. Термин происходит от слова «грок», введенного Робертом Хайнлайном в его романе «Незнакомец в чужой стране». – *Прим. ред.*

Например, если бы мы пытались построить модель для прогнозирования того, является ли животное собакой или кошкой, мы могли бы взглянуть на двумерный график высоты и веса известных собак или кошек и нарисовать линию, разделяющую два класса, как показано на рис. 10.4.

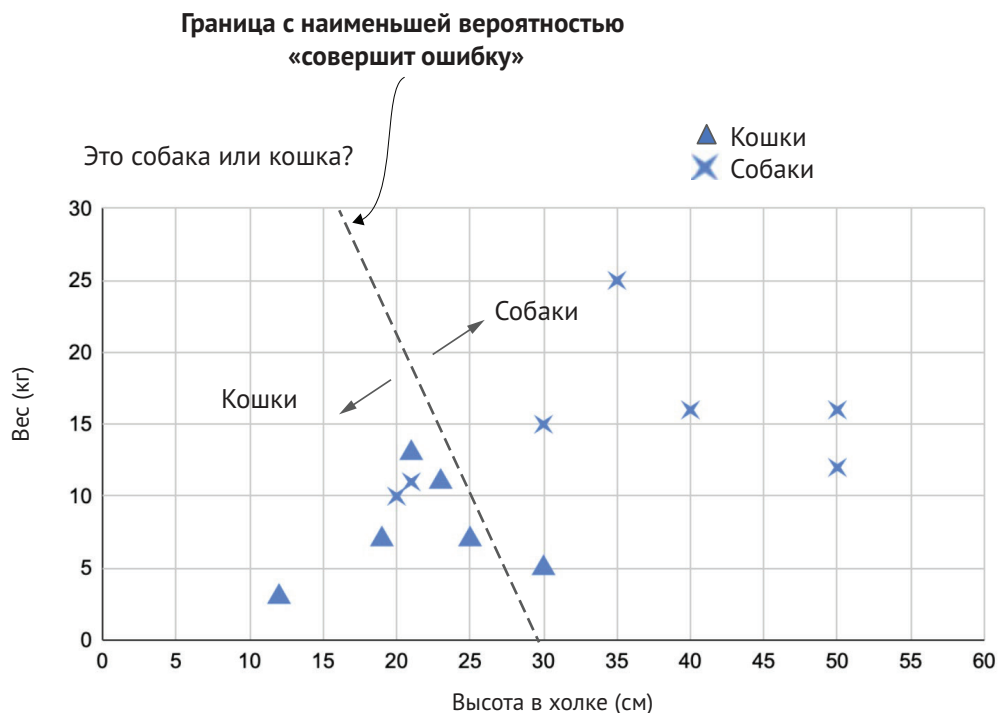


Рис. 10.4. Пример SVM: животное – собака или кошка? Эта гиперплоскость (линия здесь) разделяет эти два случая на основе двух признаков: роста и веса. Скоро вы увидите, как мы могли бы сделать что-то похожее, чтобы разделить релевантные и нерелевантные результаты поиска для запроса

Хорошая разделительная гиперплоскость, проведенная между классами, пытается минимизировать ошибки, которые она допускает при классификации обучающих данных (меньше собак на стороне кошек и наоборот). Мы также хотим, чтобы гиперплоскость была *обобщаемой*, т. е. она должна справиться с классификацией животных, которые не участвовали в процессе обучения. В конце концов, какой смысл в модели, если она не может делать прогнозы о новых данных? Она не была бы очень ИИ-поддерживаемой!

Еще одна деталь, которую нужно знать о SVM, заключается в том, что они могут быть чувствительны к диапазону наших признаков. Например, представьте, что признак height был бы миллиметрами, а не сантиметрами, как на рис. 10.5. Это заставляет данные растягиваться по оси x , и разделяющая гиперплоскость выглядит совсем иначе!

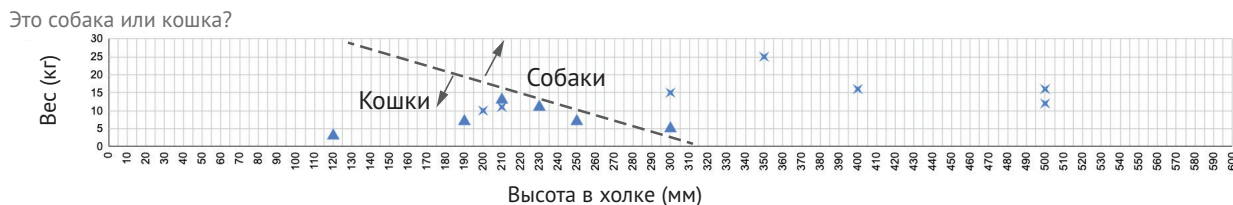


Рис. 10.5. Разделение гиперплоскости, на которую влияет диапазон одного из признаков. Это приводит к тому, что SVM становятся чувствительными к диапазону признаков, и поэтому нам нужно нормировать признаки, чтобы один признак не создавал чрезмерного влияния на модель

SVM работают лучше всего, когда наши данные нормированы. *Нормирование* просто означает масштабирование признаков до сопоставимого диапазона. Мы нормируем наши данные, сопоставив 0 со средним значением признаков. Если средний `release_year` равен 1990, фильмы, выпущенные в 1990 году, будут нормированы до 0. Мы также сопоставим +1 и -1 с одним стандартным отклонением выше или ниже среднего. Таким образом, если стандартное отклонение годов выпуска фильмов составляет 22 года, то фильмы 2012 года превращаются в 1.0; фильмы 1968 года превращаются в -1.0. Мы можем повторить это для `title_bm25` и `overview_bm25`, используя средние значения и стандартные отклонения этих признаков в наших обучающих данных. Это помогает сделать признаки немного более сопоставимыми при поиске разделяющей гиперплоскости.

После этой краткой предыстории давайте теперь рассмотрим, как SVMrank создает обобщаемую модель для различения релевантных и нерелевантных документов, и в том числе для запросов, которые он никогда раньше не видел.

10.4.2. Преобразование нашей задачи обучения LTR в бинарную классификацию

С LTR мы должны переформулировать задачу из ранжирования в традиционную задачу машинного обучения. В этом разделе мы рассмотрим, как SVMrank преобразует ранжирование в задачу бинарной классификации, подходящую для SVM.

Прежде чем начать, давайте проверим полностью логированный обучающий набор из конца шага 2 для наших двух любимых запросов, `star wars` и `social network`. В этом разделе мы сосредоточимся только на двух признаках (`title_bm25` и `overview_bm25`), чтобы помочь нам графически исследовать взаимосвязи признаков. На рис. 10.6 показаны эти два признака для каждого оцененного документа из запросов `star wars` и `social network` с маркировкой некоторых известных фильмов из обучающего набора.

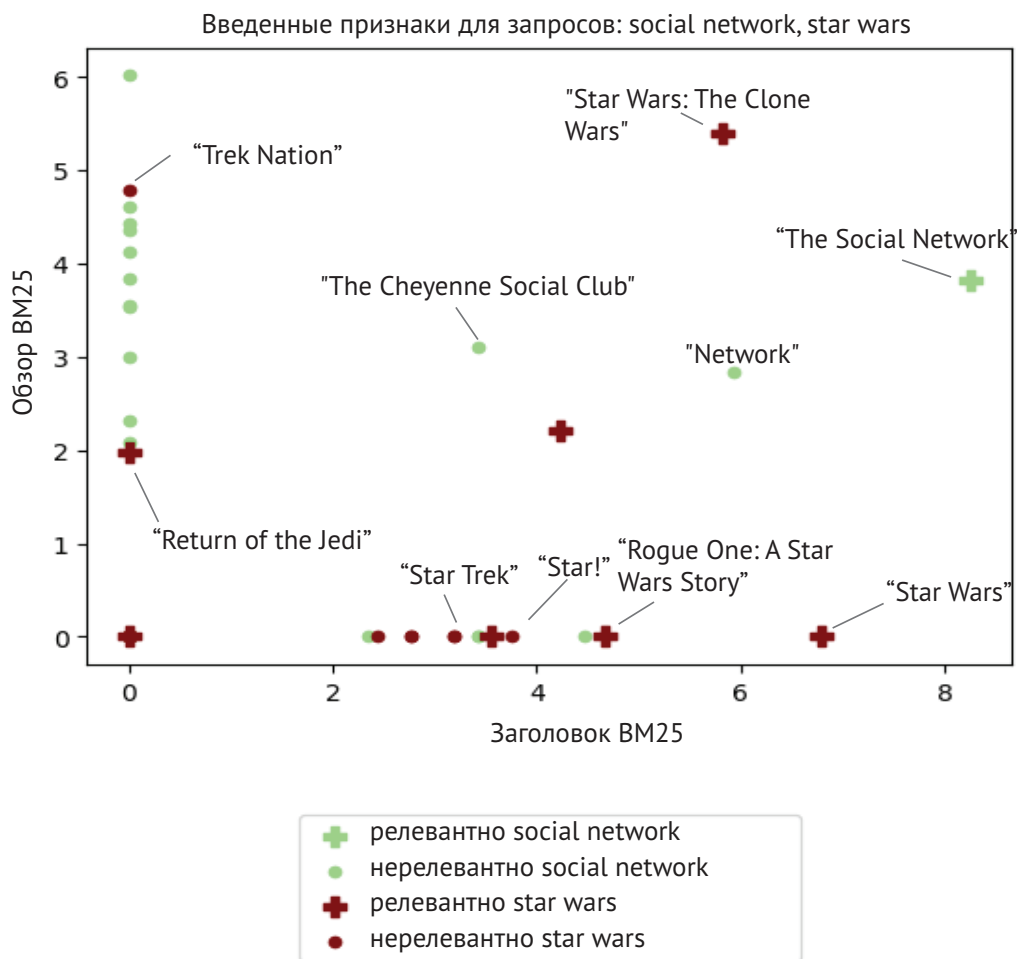


Рис. 10.6. Логированные оценки признаков для запросов social network и star wars

В первую очередь нормируйте признаки LTR

Наш первый шаг – нормировать каждый признак. Следующий листинг берет логированный вывод из шага 2 и нормирует признаки в `normed_judgments`.

Листинг 10.8. Нормирование логированных обучающих данных LTR

```
means, std_devs, normed_judgments = normalize_features(logged_judgments)
print(logged_judgments [360])
print(normed_judgments [360])
```

Вывод:

```
#Judgment(grade, keywords, doc_id,
#          qid, features, weight)
Judgment(1, "social network", 37799,
          11, [8.244, 3.814, 2010.0], 1)
Judgment(1, "social network", 37799,
          11, [4.483, 2.100, 0.835], 1)
```

Ненормированный пример
с необработанными `title_bm25`,
`overview_bm25` и `release_year`.

То же суждение, но
нормированное.

Вы можете видеть, что вывод из листинга 10.8 сначала показывает логированные оценки BM25 для названия и обзора (8.244, 3.814)

вместе с годом выпуска (2010). Затем эти признаки нормируются, где 8.244 для title_bm25 соответствует 4.483 стандартных отклонений выше среднего title_bm25 и т. д. для каждого признака.

Мы изобразили нормированные признаки на рис. 10.7. Это выглядит очень похоже на рис. 10.6, отличается только масштаб по каждой оси.

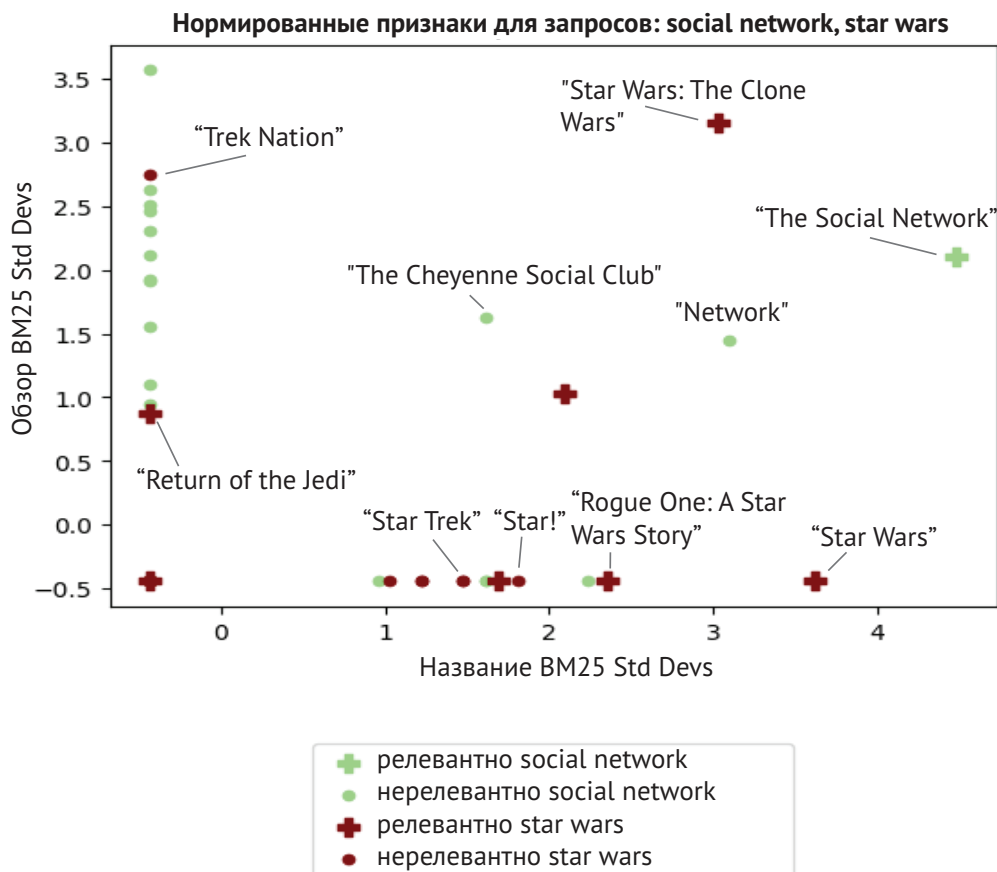


Рис. 10.7. Нормированные фильмы «Звездные войны» и «Социальная сеть». Каждое приращение на графике – это стандартное отклонение выше или ниже среднего

Далее мы превратим ранжирование в задачу обучения бинарной классификации, чтобы отделить релевантные результаты от нерелевантных.

Во-вторых, вычислите парные разницы

С нормированными данными мы заставили признаки соответствовать диапазону. Теперь наш SVM не должен быть смещен признаками, которые имеют очень большие диапазоны. В этом разделе мы готовы преобразовать задачу в задачу бинарной классификации, подготавливая почву для обучения нашей модели.

SVMrank использует парное преобразование, чтобы переформулировать LTR в задачу бинарной классификации. *Попарное тестирование*¹

¹ Попарное тестирование (англ. *pairwise testing*) – это метод тестирования, при котором каждая возможная комбинация значений параметров приложения тестируется вместе с каждой другой возможной комбинацией. При этом каждый параметр приложения используется хотя бы один раз. Этот метод позволяет увеличить эффективность тестирования и обеспечить максимально полное покрытие всех возможных комбинаций значений параметров приложения. – Прим. ред.

просто означает превращение ранжирования в задачу минимизации неупорядоченных пар для запроса.

В оставшейся части этого раздела мы тщательно рассмотрим парный алгоритм SVMrank, описанный в листинге 10.9. Алгоритм SVMrank берет каждое суждение для каждого запроса и сравнивает его с каждым другим суждением для того же запроса. Он вычисляет различия признаков (*feature_deltas*) между каждой релевантной и нерелевантной парой для этого запроса. При добавлении к *feature_deltas*, если первое суждение более релевантно, чем второе, оно помечается как +1 в *predictor_deltas*. Если первое суждение менее релевантно, оно помечается как -1. Этот алгоритм парного преобразования выдает обучающие данные (*feature_deltas* и *predictor_deltas*), необходимые для бинарной классификации.

Листинг 10.9. Преобразование признаков в парные данные для SVMrank

```
for doc1_judgment in query_judgments:
    for doc2_judgment in query_judgments:
        j1_features = numpy.array(doc1_judgment.features)
        j2_features = numpy.array(doc2_judgment.features)

        if doc1_judgment.grade > doc2_judgment.grade:
            predictor_deltas.append(+1)
            feature_deltas.append(j1_features - j2_features)
        elif doc1_judgment.grade < doc2_judgment.grade:
            predictor_deltas.append(-1)
            feature_deltas.append(j1_features - j2_features)
```

Сохраняет метку +1, если doc1 более релевантен, чем doc2.

Сохраняет дельты признаков.

Сохраняет метку -1, если doc1 менее релевантен, чем doc2.

Сохраняет дельты признаков.

На рис. 10.8 показаны попарные различия и выделены важные моменты.

Вы видите, что положительные парные дельты (+) имеют тенденцию быть ближе к правому верхнему углу. Это означает, что релевантные документы имеют более высокие *title_bm25* и *overview_bm25* по сравнению с нерелевантными.

Это много для усвоения! Давайте внимательно рассмотрим несколько примеров, шаг за шагом, чтобы увидеть, как этот алгоритм строит точки данных на рис. 10.9. Этот алгоритм сравнивает релевантные и нерелевантные документы для каждого запроса, сравнивая два документа («Network» и «The Social Network») в запросе *social network*, как показано на рис. 10.9.

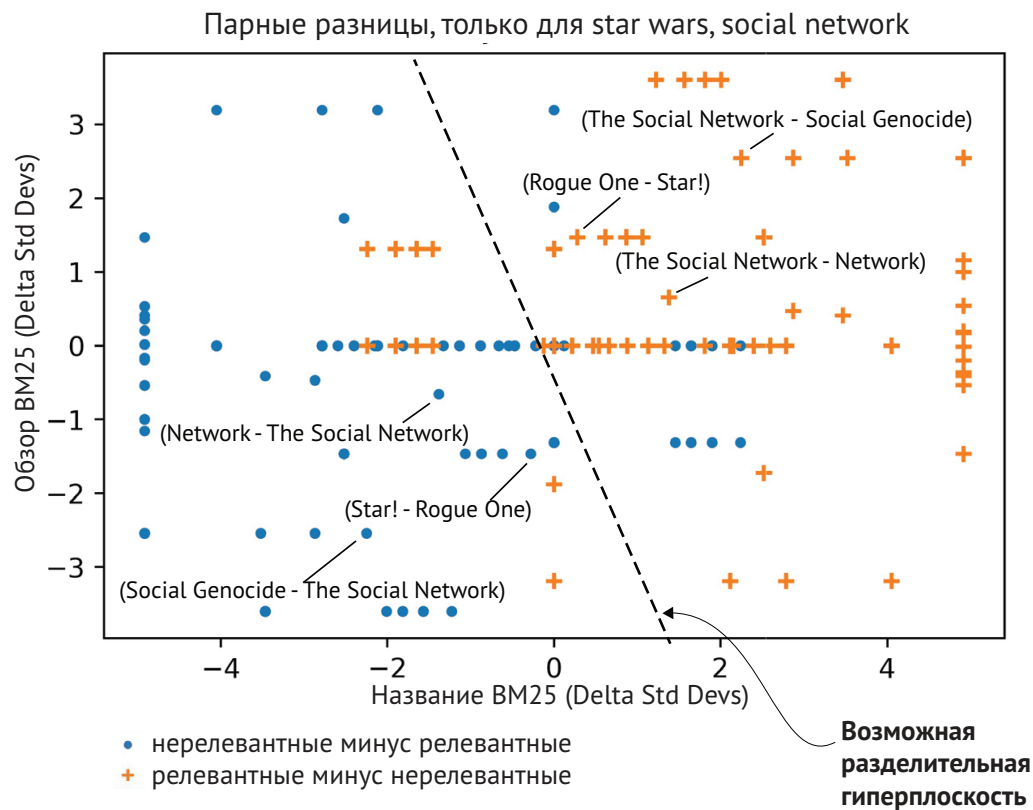


Рис. 10.8. Попарные различия после преобразования SVMrank для документов «Социальной сети» и «Звездных войн», а также гиперплоскость, разделяющая кандидатов

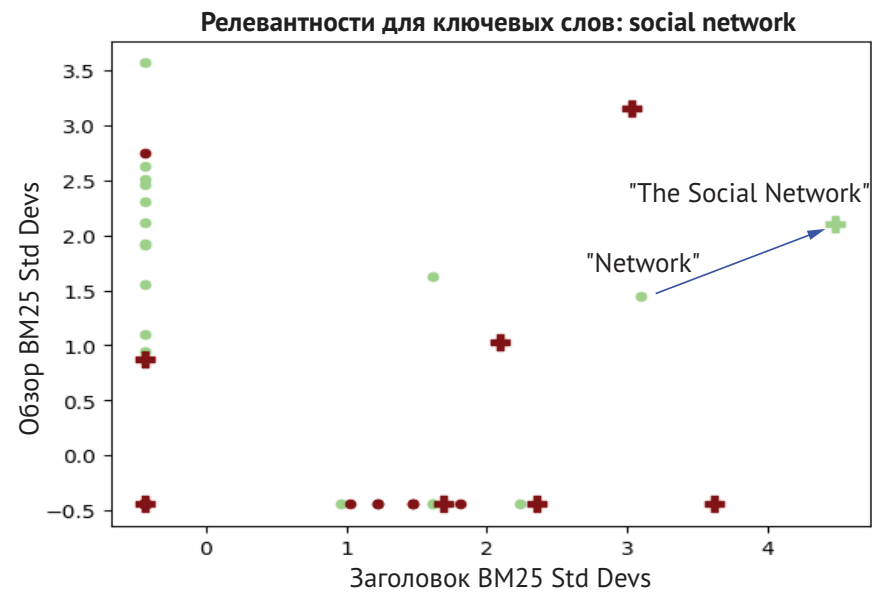


Рис. 10.9. Сравнение «Network» с «The Social Network» для запроса social network

Вот признаки для «The Social Network»:

#[title_bm25, overview_bm25] [4.483, 2.100]		title_bm25 на 4.483 стандартных отклонений выше среднего, а overview_bm25 на 2.100 стандартных отклонений выше среднего.
--	--	---

Вот признаки для «Network»:

```
#[title_bm25, overview_bm25]
[3.101, 1.443]
```

title_bm25 на 3.101 стандартных отклонений выше среднего, а overview_bm25 на 1.443 стандартных отклонений выше среднего.

Затем мы вставляем дельту между «The Social Network» и «Network» в следующий листинг.

Листинг 10.10. Расчет и сохранение дельты признака

```
predictor_deltas.append(+1)
feature_deltas.append([4.483, 2.100] - [3.101, 1.443])
```

←
Добавляет [1.382,
0.657] к feature_deltas.

Переформулируя листинг 10.10, можно сказать, что вот один пример фильма «The Social Network», который более релевантен, чем фильм «Network» для этого запроса `social network`. Интересно! Давайте посмотрим, что их отличает. Конечно, «разница» в математике означает вычитание, что мы и сделаем здесь. Ах да, после вычитания разницы мы видим, что `title_bm25` у «The Social Network» на 1.382 стандартных отклонений выше, чем у «Network»; аналогично `overview_bm25` на 0.657 стандартных отклонений выше. Действительно, обратите внимание на + для «The Social Network» минус «Network» на рис. 10.8, показывающий точку [1.382, 0.657] среди дельт.

Алгоритм также отметил бы, что «Network» менее релевантен, чем «» для запроса `social network`, как показано на рис. 10.10.

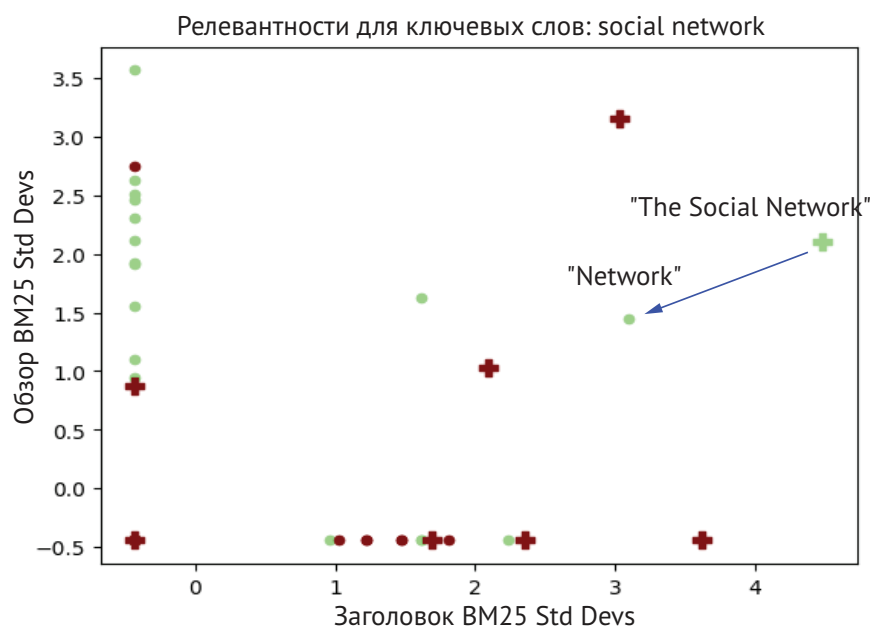


Рис. 10.10. Сравнение «Network» с «The Social Network» для запроса `social network`

Как и в листинге 10.9, наш код фиксирует эту разницу в релевантности между этими двумя документами, но на этот раз в противоположном направлении (нерелевантный-минус-релевантный). Поэтому неудивительно, что мы видим те же значения, но наоборот.

```
predictor_deltas.append(-1)
feature_deltas.append([3.101, 1.443] - [4.483, 2.100])
```

Оценивается как
[-1.382, -0.657]

На рис. 10.11 мы переходим к другому релевантно-нерелевантному сравнению двух документов для запроса `social network`, добавляя еще одно сравнение к новому обучающему набору.

В листинге 10.11 показано добавление как положительных дельт (с более релевантным документом, перечисленным первым), так и отрицательных дельт (с менее релевантным документом, перечисленным первым) для выделенной пары документов, сравниваемых на рис. 10.11.

Листинг 10.11. Сложение положительных и отрицательных дельт

```
# Positive example
predictor_deltas.append(+1)
feature_deltas.append([4.483, 2.100] - [2.234, -0.444])
```

Оценивается как
[2.249, 2.544].

```
# Negative example
predictor_deltas.append(-1)
feature_deltas.append([2.234, -0.444] - [4.483, 2.100])
```

Оценивается как
[-2.249, -2.544].

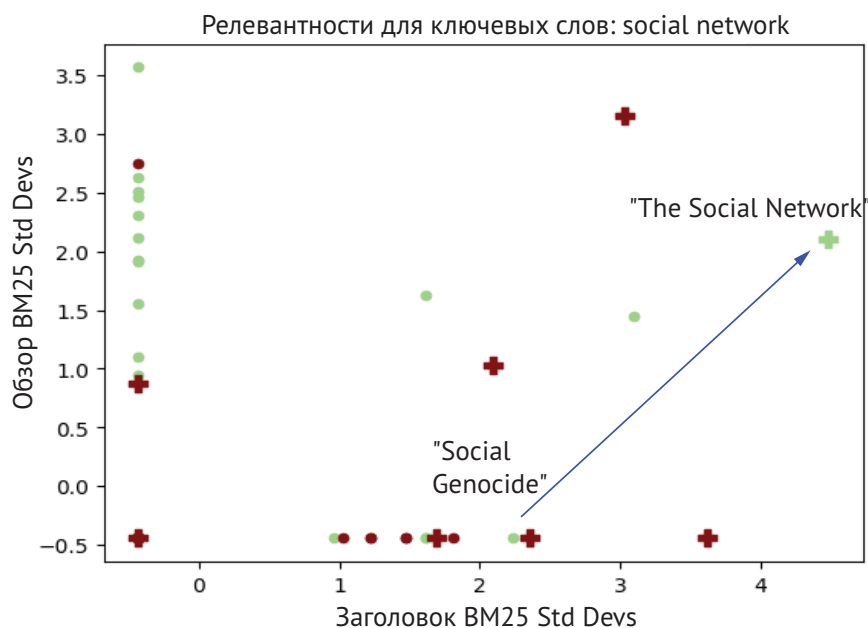


Рис. 10.11. Сравнение «Social Genocide» с «The Social Network» для запроса `social network`

После того как мы переберем все парные различия между документами, соответствующими запросу *social network*, чтобы создать точечный обучающий набор, мы можем перейти к регистрации различий для других запросов. Рисунок 10.12 показывает различия для второго запроса, на этот раз сравнивая релевантность документов, соответствующих запросу *star wars*.



Рис. 10.12. Сравнение «Rogue One: A Star Wars Movie» со «Star!» для запроса *star wars*. Мы отошли от *social network* и начали рассматривать закономерности в другом запросе.

# Positive example	Признаки «Rogue One»
<code>predictor_deltas.append(+1)</code>	минус признаки «Star!».
<code>feature_deltas.append([2.088, 1.024] - [1.808, -0.444])</code>	←
# Negative example	Признаки «Star!» минус
<code>predictor_deltas.append(-1)</code>	признаки «Rogue One».
<code>feature_deltas.append([1.808, -0.444] - [2.088, 1.024])</code>	←

Мы продолжаем этот процесс вычисления различий между значениями признаков для релевантных и нерелевантных документов, пока не вычислим все парные различия для наших обучающих и тестовых запросов.

Как вы можете увидеть на рис. 10.8, положительные примеры показывают положительную дельту `title_bm25` и, возможно, слегка положительную дельту `overview_bm25`. Это станет еще более очевидным, если мы вычислим дельты по полному набору данных из 100 запросов, как показано на рис. 10.13.

Интересно! Теперь очень легко визуально определить, что большее совпадение оценок `title_bm25` сильно коррелирует с тем, что документ релевантен запросу, и что более высокая оценка `overview_bm25` тоже положительно коррелирует, но немного меньше.

Стоит сделать шаг назад и спросить, подходит ли эта формулировка ранжирования для вашей области. Различные модели LTR имеют свой собственный метод преобразования парных сравнений в задачи классификации по мере необходимости. В качестве другого примера LambdaMART – популярный алгоритм LTR, основанный на усиленных деревьях – использует парный обмен и измеряет изменение *дисконтированного кумулятивного выигрыша*¹.

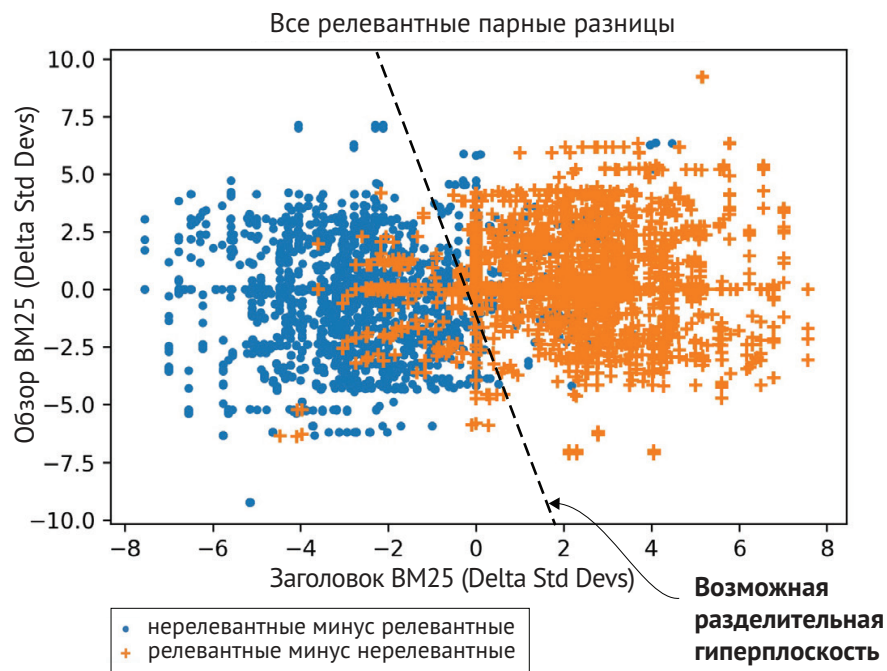


Рис. 10.13. Полный обучающий набор с гиперплоскостью, разделяющей релевантные и нерелевантные документы. Мы видим закономерность! Релевантные документы имеют более высокий `title_bm25` и, возможно, менее высокий `overview_bm25`

Далее мы обучим надежную модель, чтобы зафиксировать закономерности в нашем полностью преобразованном наборе данных ранжирования.

10.5. Шаг 4: обучение (и тестирование!) модели

Хорошее машинное обучение, очевидно, требует большой подготовки данных. К счастью, вы добрались до раздела, где мы фактически обучаем модель! С `feature_deltas` и `predictor_deltas` из последнего раздела у нас теперь есть обучающий набор, подходящий для обучения

¹ Дисконтированный кумулятивный выигрыш (англ. *Discounted cumulative gain, DCG*) – это метрика качества ранжирования в поиске информации. Она суммирует полезность результатов с учетом их позиции в списке результатов. В информатике ее используют для оценки эффективности алгоритмов веб-поисковых систем или связанных приложений. DCG измеряет полезность документа в зависимости от его позиции в списке результатов. Для расчета релевантность элемента умножают на вес, равный обратному логарифму номера позиции. – Прим. ред.

классификатора ранжирования. Эта модель позволит нам предсказывать, когда документы могут быть релевантными, даже для запросов и документов, которые она еще не видела.

10.5.1. Превращение вектора разделяющей гиперплоскости в оценочную функцию

Мы увидели, как разделяющая гиперплоскость SVMrank может классифицировать и отличать нерелевантные примеры от релевантных. Это полезно, но вы, возможно, помните, что наша задача – найти *оптимальные* веса для наших признаков, а не только классифицировать документы. Поэтому давайте рассмотрим, как мы можем оценивать *результаты* поиска с помощью этой гиперплоскости.

Оказывается, разделяющая гиперплоскость также дает нам то, что нам нужно для обучения оптимальным весам. Любая гиперплоскость определяется вектором, ортогональным к плоскости. Поэтому, когда библиотека машинного обучения SVM выполняет свою работу, она дает нам представление о весах, которые должен иметь каждый признак, как показано на рис. 10.14.

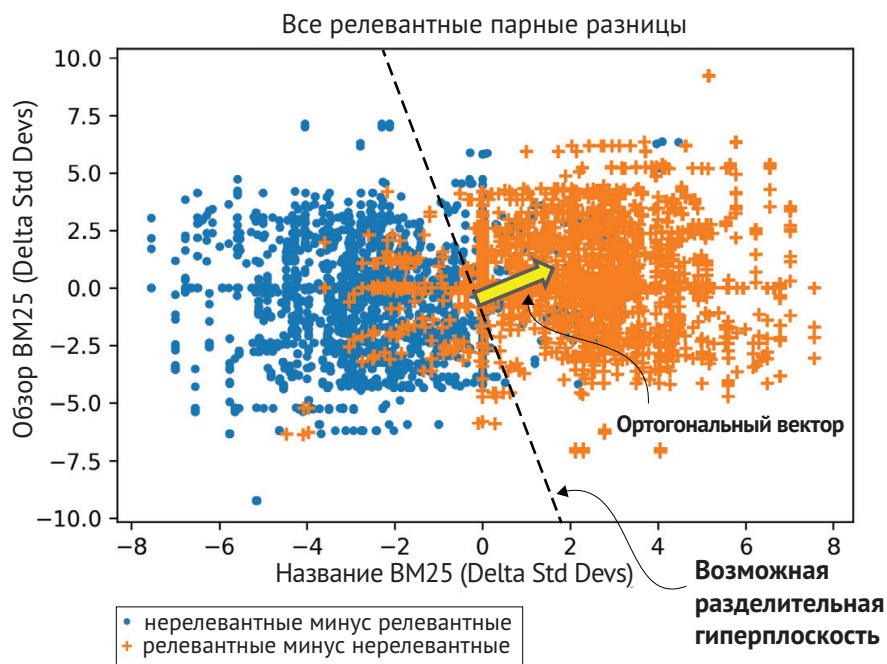


Рис. 10.14. Полный обучающий набор с гиперплоскостью, разделяющей кандидатов, показывающий ортогональный вектор, определяющий гиперплоскость

Подумайте, что представляет собой этот ортогональный вектор. Этот вектор указывает в направлении релевантности! Он говорит, что релевантные примеры находятся в этом направлении, а нерелевантные – в противоположном. Этот вектор *определенно* указывает на то, что `title_bm25` оказывает сильное влияние на релевантность, а `review_bm25` оказывает меньшее влияние. Этот вектор может быть примерно таким:

[0.65, 0.40]

Мы использовали алгоритм парного преобразования в листинге 10.9 для вычисления дельт, необходимых для выполнения классификации между нерелевантными и релевантными примерами. Если мы обучим SVM на этих данных, как в следующем листинге, модель даст нам вектор, определяющий разделяющую гиперплоскость.

Листинг 10.12. Обучение линейного SVM с помощью scikit-learn

```
from sklearn import svm
model = svm.LinearSVC(max_iter=10000)
model.fit(feature_deltas, predictor_deltas)
display(model.coef_)

Вывод:
array([0.40512169, 0.29006328, 0.14451715])
```

Создает линейную модель с помощью sklearn.

Подгоняет к дельтам с помощью SVM.

Вектор, который определяет разделяющую гиперплоскость.

Листинг 10.12 обучает SVM разделять `predictor_deltas` (помните, что они +1 и -1) с помощью соответствующих `feature_deltas` (дельты в нормированных признаках `title_bm25`, `overview_bm25` и `release_year`). Результирующая модель представляет собой вектор, ортогональный разделяющей гиперплоскости. Как и ожидалось, он показывает сильный вес на `title_bm25`, более скромный на `overview_bm25` и более слабый вес на `release_year`.

10.5.2. Тест-драйв модели

Как эта модель работает как функция ранжирования? Предположим, пользователь вводит запрос `wrath of khan`. Как эта модель может оценить документ «Star Trek II: The Wrath of Khan» относительно этого запроса? Ненормированный вектор признаков указывает на сильное соответствие заголовка и обзора для этого запроса.

```
[5.9217176, 3.401492, 1982.0]
```

Необработанные признаки для «Star Trek II».

При его нормировании каждое значение признака составляет такое количество стандартных отклонений выше или ниже среднего значения каждого признака:

```
[3.099, 1.825, -0.568]
```

Нормированные признаки для «Star Trek II».

Мы просто умножаем каждый нормированный признак на его соответствующее значение `coef_`. Суммируя их, мы получаем оценку релевантности:

```
(3.099 * 0.405) + (1.825 * 0.290) + (-0.568 * 0.1445) = 1.702
```

Расчет оценки релевантности для «Star Trek II».

Как эта модель оценила бы «Star Trek III: The Search for Spock» относительно «Star Trek II: The Wrath of Khan» для нашего запроса `wrath of khan`? Надеюсь, не так высоко! Действительно, это не так:

$$\begin{array}{l}
 [0.0, 0.0, 1984.0] \quad \leftarrow \text{Необработанные признаки для «Star Trek III»} \\
 [-0.432, -0.444, -0.468] \quad \leftarrow \text{Нормированные признаки для «Star Trek III»} \\
 (-0.432 * 0.405) + (-0.444 * 0.290) + (-0.468 * 0.1445) = -0.371 \quad \leftarrow \text{Расчет релевантности для «Star Trek III»}
 \end{array}$$

Кажется, модель правильно предсказывает наиболее релевантный ответ.

10.5.3. Валидация модели

Тестирование пары запросов помогает нам выявлять проблемы, но мы бы предпочли более систематический способ проверки того, является ли модель обобщаемой.

Одно из отличий LTR от традиционного машинного обучения заключается в том, что мы обычно оцениваем запросы и целые наборы результатов, а не отдельные точки данных, чтобы доказать эффективность нашей модели. Мы выполним разделение теста и обучения на уровне запроса. Это позволит нам выявить типы запросов с проблемами. Мы оценим с помощью простой метрики точности, подсчитав долю результатов в верхнем K (с $k=5$ в нашем случае), которые являются релевантными. Вам следует выбрать метрику релевантности, наиболее подходящую для вашего собственного варианта использования.

Сначала мы случайным образом поместим наши запросы в тестовый или обучающий набор, как показано в следующем листинге.

Листинг 10.13. Простое разделение теста и обучения на уровне запроса

```

all_qids = list(set([j.qid for j in normed_judgments]))
random.shuffle(all_qids)
proportion_train = 0.1

split_idx = int(len(all_qids) * proportion_train)
test_qids = all_qids[:split_idx]
train_qids = all_qids[split_idx:]

train_data = []; test_data=[]
for j in normed_judgments:
    if j.qid in train_qids:
        train_data.append(j)
    elif j.qid in test_qids:

```

Определяет случайные 10 % суждений для включения в обучающий набор.

Помещает каждое суждение в обучающие данные (10 %) или в тестовый набор (90 %).

Разделив обучающие данные, мы можем выполнить трюк парного преобразования из шага 3. Затем мы можем повторно обучиться только на обучающих данных.

Листинг 10.14. Обучение только на обучающих данных

```
train_data_features, train_data_predictors = pairwise_transform(train_data)
```

```
from sklearn import svm
model = svm.LinearSVC(max_iter=10000, verbose=1)
model.fit(train_data_features, train_data_predictors)
display(model.coef_[0])
```

←
Подходит толь-
ко для обучаю-
щих данных.

Вывод:

```
array([0.37486809, 0.28187458, 0.12097921])
```

До сих пор мы сдерживали данные тестов. Как хороший учитель, мы не хотим давать ученику все ответы. Мы хотим посмотреть, узнала ли модель что-нибудь, кроме механического запоминания обучающих примеров.

В следующем листинге мы оцениваем нашу модель, используя тестовые данные. Этот код проходит по каждому тестовому запросу и ранжирует каждое тестовое суждение, используя модель. Затем он вычисляет точность для четырех лучших суждений.

Листинг 10.15. Может ли наша модель обобщать за пределами обучающих данных?

```
def score_one(features, model):
    score = 0.0
    for idx, f in enumerate(features):
        this_coef = model.coef_[0][idx].item()
        score += f * this_coef
    return score

def rank(query_judgments, model):
    for j in query_judgments:
        j.score = score_one(j.features, model)
    return sorted(query_judgments, key=lambda j: j.score, reverse=True)

def evaluate_model(test_data, model, k=5):
    total_precision = 0
    unique_queries = groupby(test_data, lambda j: j.qid)
    num_groups = 0
    for qid, query_judgments in unique_queries:
        num_groups += 1
        ranked = rank(list(query_judgments), model)
        total_relevant = len([j for j in ranked[:k]
                              if j.grade == 1])
        total_precision += total_relevant / float(k)
    return total_precision / num_groups

evaluation = evaluate_model(test_data, model)
print(evaluation)
```

← Для каждого те-
стового запроса.

← Вычисляет точность
для этого запроса.

← Оценивает каж-
дое суждение
и ранжирует этот
запрос, исполь-
зуя модель.

Оценка:

0.36

При нескольких запусках следует ожидать точность примерно 0.3-0.4. Неплохо для нашей первой итерации, в которой мы просто угадали несколько признаков (`title_bm25`, `overview_bm25` и `release_year`)!

В LTR вы всегда можете вернуться к предыдущим шагам, чтобы увидеть, что можно улучшить. Этот тест точности – первый раз, когда мы смогли систематически оценить нашу модель, поэтому сейчас самое время пересмотреть признаки, чтобы увидеть, как точность может быть улучшена в последующих запусках. Вернитесь к шагу 2. Посмотрите, какие примеры находятся на неправильной стороне разделяющей гиперплоскости. Например, если вы посмотрите на рис. 10.8, третий фильм «Звездных войн» – «Возвращение джедая» соответствует шаблону соответствующего документа, в названии которого нет совпадения по ключевому слову. При отсутствии названия какие еще признаки можно добавить, чтобы помочь зафиксировать принадлежность фильма к определенной коллекции, например «Звездным войнам»? Возможно, в наборе данных TMDb есть свойство, с которым мы могли бы поэкспериментировать.

Но пока давайте возьмем только что созданную нами модель и посмотрим, как мы можем развернуть ее в производстве.

10.6. Шаги 5 и 6: загрузка модели и поиск

В этом разделе мы, наконец, загрузим нашу модель, чтобы ее можно было применить для ранжирования будущих результатов поиска. Затем обсудим как применение модели для ранжирования всех документов, так и ее применение для реранкинга уже запущенного и, вероятно, более эффективного первоначального запроса. Наконец, мы обсудим некоторые последствия использования моделей LTR для производительности.

10.6.1. Запуск и использование модели LTR

Изначально мы представили нашу цель как поиск *идеальных* бустов для жестко закодированной функции ранжирования, такой как в листинге 10.2:

```
{"query": f"title:({keywords})^10 overview:({keywords})^20  
➔{!func}release_year^0.01"}
```

Этот усиленный запрос действительно умножает каждый признак на вес (бустинг) и суммирует результаты. Но оказывается, что мы не хотим, чтобы поисковая система умножала *необработанные* значения признаков. Вместо этого нам нужно нормировать значения признаков.

Многие поисковые системы позволяют нам хранить линейную модель ранжирования вместе со статистикой нормирования признаков. Мы сохранили средние значения и `std_devs` каждого признака, которые будут использоваться для нормирования значений любого оцениваемого документа. Эти коэффициенты связаны с каждым признаком при загрузке модели, как показано в следующем листинге.

Листинг 10.16. Создание и загрузка линейной модели

```
model_name = "movie_model"
feature_names = ["title_bm25", "overview_bm25", "release_year"]
linear_model = ltr.generate_model(model_name, feature_names,
                                  means, std_devs, model.coef_[0])
response = ltr.upload_model(linear_model)
display(linear_model)
```

Сгенерированная линейная модель (для `engine=solr`):

```
{
  "store": "movies",
  "class": "org.apache.solr.ltr.model.LinearModel",
  "name": "movie_model",
  "features": [
    {
      "name": "title_bm25",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "0.7245440735518126",
          "std": "1.6772600303613545"
        }
      },
      "weights": {
        "title_bm25": 0.3748679655554891,
        "overview_bm25": 0.28187459845467566,
        "release_year": 0.12097924576841014
      }
    },
    {
      "name": "overview_bm25",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "0.6662927508611409",
          "std": "1.4990448120673643"
        }
      },
      "weights": {
        "title_bm25": 0.3748679655554891,
        "overview_bm25": 0.28187459845467566,
        "release_year": 0.12097924576841014
      }
    },
    {
      "name": "release_year",
      "norm": {
        "class": "org.apache.solr.ltr.norm.StandardNormalizer",
        "params": {
          "avg": "1993.3349740932642",
          "std": "19.964916628520722"
        }
      },
      "weights": {
        "title_bm25": 0.3748679655554891,
        "overview_bm25": 0.28187459845467566,
        "release_year": 0.12097924576841014
      }
    }
  ],
  "params": {
    "weights": {
      "title_bm25": 0.3748679655554891,
      "overview_bm25": 0.28187459845467566,
      "release_year": 0.12097924576841014
    }
  }
}
```

Хранилище признаков для поиска признаков.

Какой признак выполнить перед оценкой этой модели.

Как нормировать этот признак перед применением веса.

Вес каждого признака в модели.

Ответ из листинга 10.16 Solr-специфичен и будет меняться в зависимости от того, какую поисковую систему вы настроили. Затем мы можем выполнить поиск, используя загруженную модель LTR, как показано в следующем листинге.

Листинг 10.17. Ранжирование всех документов с помощью модели LTR для `harry potter`

```
request = {"query_fields": ["title", "overview"],
          "return_fields": ["title", "id", "score"],
```

```

    "rerank_query": "harry potter",
    "log": True}
response = ltr.search_with_model("movie_model", **request)
display(response["docs"])

```

Движок-специфичный поисковый запрос (для engine=solr):

```

{"fields": ["title", "id", "score"],
 "limit": 5,
 "query": "{!ltr reRankDocs=9999999
           ➔model=movie_model efi.keywords=\"harry potter\"}"}

```

Выполняет нашу модель по максимальному количеству документов с указанными параметрами.

Возвращенные документы:

```

[{"id": "570724", "title": "The Story of Harry Potter", "score": 2.4261155},
 {"id": "116972", "title": "Discovering the Real World of Harry Potter",
  "score": 2.247846},
 {"id": "672", "title": "Harry Potter and the Chamber of Secrets",
  "score": 2.017499},
 {"id": "671", "title": "Harry Potter and the Philosopher's Stone",
  "score": 1.9944705},
 {"id": "54507", "title": "A Very Potter Musical",
  "score": 1.9833609}]

```

В листинге 10.17 модель LTR ранжирует все документы в корпусе, используя ключевые слова в параметре `rerank_query` в качестве входных данных для модели. Поскольку в запросе не указан начальный параметр запроса, к коллекции не применяется фильтр сопоставления до того, как результаты поиска (все документы) будут ранжированы моделью LTR. Хотя оценка такого большого количества документов с помощью модели приведет к нетривиальной задержке, это позволяет нам протестировать модель напрямую, без каких-либо других параметров сопоставления.

Обратите внимание на использование в листинге 10.17 термина «rerank» в параметре `rerank_query`. Как подразумевает этот термин, LTR обычно применяется как вторая фаза ранжирования результатов, сначала рассчитанных более эффективным алгоритмом, таким как BM25 или алгоритмом начального булева соответствия¹. Это необходимо для уменьшения количества документов, которые должны быть оценены более дорогой моделью LTR. Следующий листинг демонстрирует выполнение базового поиска и последующий реранкинг 500 лучших результатов с помощью модели LTR.

¹ Boolean match в программировании – это оператор, который сопоставляет шаблон с выражением и возвращает булево значение. – *Прим. ред.*

Листинг 10.18. Поиск для harry potter и реранкинг с помощью модели

```
request = {"query": "harry potter",
          "query_fields": ["title", "overview"],
          "return_fields": ["title", "id", "score"],
          "rerank_query": "harry potter",
          "rerank_count": 500,
          "log": True}
response = ltr.search_with_model("movie_model", **request)
display(response["docs"])
```

Движок-специфичный поиск (для engine=solr):

```
{"query": "harry potter",
 "fields": ["title", "id", "score"],
 "limit": 5,
 "params": {
   "rq": "{!ltr reRankDocs=500 model=movie_model
   ➔efi.keywords=\"harry potter\"}",
   "qf": ["title", "overview"],
   "defType": "edismax"}}
```

Первый проход Solr-запроса – простой запрос по ключевым словам с ранжированием BM25.

Переранжирует только 500 лучших документов.

Возвращенные документы:

```
[{"id": "570724", "title": "The Story of Harry Potter", "score": 2.4261155},
 {"id": "116972", "title": "Discovering the Real World of Harry Potter",
  "score": 2.247846},
 {"id": "672", "title": "Harry Potter and the Chamber of Secrets",
  "score": 2.017499},
 {"id": "671", "title": "Harry Potter and the Philosopher's Stone",
  "score": 1.9944705},
 {"id": "54507", "title": "A Very Potter Musical", "score": 1.9833605}]
```

Этот запрос намного быстрее, и он по-прежнему выдает те же самые лучшие результаты при выполнении более дешевого начального ранжирования BM25 по отфильтрованному query с последующим более дорогим реранкингом на основе LTR только по 500 лучшим результатам.

10.6.2. Примечание о производительности LTR

Как вы видите, для построения реальной модели LTR требуется много шагов. Давайте завершим главу некоторыми дополнительными мыслями о практических ограничениях производительности в системах LTR.

- Сложность модели – чем сложнее модель, тем она может быть точнее. Более простая модель может быть быстрее и проще для понимания, хотя, возможно, и менее точна. Здесь мы остановились

на очень простой модели (наборе линейных весов). Представьте себе сложную модель глубокого обучения – насколько хорошо она будет работать? Будет ли сложность оправдана? Будет ли она такой же обобщаемой (или может быть более обобщаемой)?

- Глубина реранкинга – чем глубже вы делаете реранкинг, тем больше вы можете найти дополнительных документов, которые могут оказаться скрытыми драгоценными камнями. С другой стороны, чем глубже вы делаете реранкинг, тем больше вычислительных циклов ваша модель тратит на оценку результатов в вашем кластере поисковой системы в реальном времени.
- Сложность признаков – если вы вычисляете очень сложные признаки во время запроса, они могут помочь вашей модели. Однако они замедляют оценку и время отклика поиска.
- Количество признаков – модель со многими признаками может привести к более высокой релевантности. Однако также требуется больше времени для вычисления каждого признака в каждом документе, поэтому спросите себя, какие признаки имеют решающее значение. Многие академические системы LTR используют сотни. Практические системы LTR обычно сводят их к десяткам. Вы почти всегда будете видеть убывающую отдачу от ранжирования релевантности и рост затрат на вычисления и задержки по мере добавления дополнительных признаков, поэтому важно расставить приоритеты относительно того, какие признаки включать.

Кросс-кодировщики

Кросс-кодировщик – это специализированный вид модели ранжирования с машинным обучением. Кросс-кодировщики обучены оценивать релевантность двух частей ввода (обычно текста), таких как запрос и документ. Они используют архитектуру трансформера для объединения обеих частей ввода в единое представление, которое затем используется в поиске для ранжирования релевантности документа для запроса на основе интерпретации как запроса, так и документа в их общем семантическом контексте. Перекрестные кодировщики являются классификаторами ранжирования, как и другие модели LTR, но уникальны тем, что они предварительно обучены на большом количестве данных и, как правило, сосредоточены только на текстовом сходстве между запросом и документом, а не на других признаках, таких как популярность, новизна или поведение пользователя. Хотя их можно точно настроить на вашем наборе данных, их часто используют как есть, поскольку они уже обучены на большом количестве данных и могут хорошо обобщаться на новые текстовые входные данные.

Перекрестные кодировщики очень просты в использовании без настройки, и они часто являются самым простым способом начать работу с машинным ранжированием без необходимости проводить собственное обучение. Перекрестные кодировщики, как правило, медленные, поэтому их обычно не используют для реранкинга большого количе-

ства документов. В этой и следующей главах мы сосредоточимся на более гибких моделях, способных использовать отраженный интеллект, включая те, которые обучены на суждениях ваших пользователей и неявных суждениях из пользовательских сигналов; но хорошо быть знакомым с кросс-кодировщиками, поскольку они являются популярным выбором для многих поисковых команд, особенно когда они только начинают свою деятельность. Мы рассмотрим кросс-кодировщики более подробно с примерами кода в разделе 13.7.

10.7. Почистить и повторить

Поздравляем! Вы выполнили один полный цикл LTR! Однако, как и во многих задачах с данными, вам, вероятно, придется продолжать итерации по задаче. Всегда есть что-то новое, что вы можете сделать для улучшения.

Во второй итерации вы можете рассмотреть следующее:

- новые и улучшенные признаки – есть ли типы запросов или примеров, на которых модель работает плохо, например поиск по title, где нет упоминания названия фильма? («Star Wars» не упоминаются в названии «Return of the Jedi». Какие признаки могли бы их охватить?) Можем ли мы включить уроки из глав 1–9 для создания более продвинутых признаков?
- охват обучающими данными всех признаков – обратная сторона большего количества признаков – больше обучающих данных. По мере увеличения признаков, которые вы хотели бы попробовать, вы должны задаться вопросом, достаточно ли в ваших обучающих данных примеров релевантных и нерелевантных документов для каждой различной комбинации ваших признаков. В противном случае ваша модель не будет знать, как использовать признаки для решения задачи;
- различные архитектуры моделей – мы использовали относительно простую модель, которая ожидает, что признаки будут линейно и независимо коррелировать с релевантностью, но релевантность часто может быть нелинейной и многомерной. Покупатель, ищущий iPad, может ожидать самую последнюю версию Apple iPad, за исключением случаев, когда он добавляет слово «кабель», делая запрос «ipad cable». Для этого запроса покупатель может просто захотеть самый дешевый кабель, который он сможет найти, а не самый последний. В этом случае могут быть признаки «новизны» и «цены», которые активируются в зависимости от определенных комбинаций ключевых слов, что требует более сложной архитектуры модели.

В следующей главе мы сосредоточимся на основе хорошего LTR: правильных суждениях!

Резюме

- Обучение ранжированию (LTR) создает обобщенные функции ранжирования, которые можно применять ко всем поискам, используя надежные методы машинного обучения.
- Признаки LTR обычно соответствуют поисковым запросам. Поисковые системы, которые поддерживают LTR, часто позволяют вам сохранять и регистрировать признаки для использования при обучении и последующем применении модели ранжирования.
- У нас есть огромная свобода в том, какие признаки мы используем для обобщения релевантности. Признаки могут быть свойствами запросов (например, количество терминов), свойствами документов (например, популярность) или отношениями между запросами и документами (например, BM25 или другие оценки релевантности).
- Чтобы хорошо выполнять LTR и применять известные методы машинного обучения, мы обычно переформулируем задачу ранжирования релевантности в традиционную задачу поточечного машинного обучения.
- SVMrank создает простые линейные веса для нормированных значений признаков, что является хорошим первым шагом на вашем пути к LTR.
- Чтобы быть по-настоящему полезной, нам нужно, чтобы наша модель обобщала за пределами того, чему она научилась. Мы можем подтвердить способность модели LTR к обобщению, отложив некоторые суждения в сторону в тестовом наборе данных и не используя их во время обучения. После обучения мы можем оценить модель на этом ранее невиданном тестовом наборе данных, чтобы подтвердить способность модели к обобщению.
- После загрузки модели LTR в вашу поисковую систему обязательно рассмотрите компромиссы производительности (скорости) с релевантностью. Реальным поисковым системам требуется и то и другое.

11

Автоматизация обучения ранжированию с помощью моделей кликов

В этой главе рассматривается:

- автоматизация переобучения ранжированию на основе поведенческих сигналов пользователя (поиски, клики и т. д.);
- преобразование сигналов пользователя в неявные данные обучения LTR с помощью моделей кликов;
- преодоление тенденции пользователя кликать на предметы выше в результатах поиска, независимо от релевантности;
- обработка документов с низкой достоверностью с меньшим количеством кликов при получении неявных суждений.

В главе 10 мы шаг за шагом прошли обучение модели обучения ранжированию (LTR). Подобно тому как мы разбирали механику сборки автомобиля, мы увидели основные гайки и болты обучения модели LTR. В этой главе мы будем рассматривать процесс обучения LTR как черный ящик. Другими словами, мы отойдем от внутренних компонентов LTR, вместо этого относясь к LTR больше как к беспилотному автомобилю, точно настраивая его поездку к конечному пункту назначения.

Вспомните, что LTR опирается на точные данные обучения, чтобы быть эффективным. Данные обучения LTR описывают, как пользователи ожидают оптимального ранжирования результатов поиска; они предоставляют указания, которые мы будем вводить в наш беспилотный автомобиль LTR. Как вы увидите, определение того, что является релевантным на основе взаимодействия с пользователем, сопряжено со многими трудностями. Однако, если мы сможем преодолеть эти трудности и обрести высокую уверенность в наших данных обучения, мы сможем построить *автоматизированное обучение ранжированию*: систему, которая регулярно переобучает LTR, чтобы фиксировать последние ожидания релевантности пользователя.

Поскольку данные обучения играют столь важную роль в автоматизированном LTR, задачи становятся не «Какую модель / функции / поисковую систему нам следует использовать?», а более фундаментальными: «Чего пользователи хотят от поиска?», «Как превратить это в данные обучения?» и «Как мы узнаем, хороши ли эти данные обучения?» Повышая нашу уверенность в ответах на эти вопросы, мы можем поставить (пере)обучение LTR на автопилот, как показано на рис. 11.1.

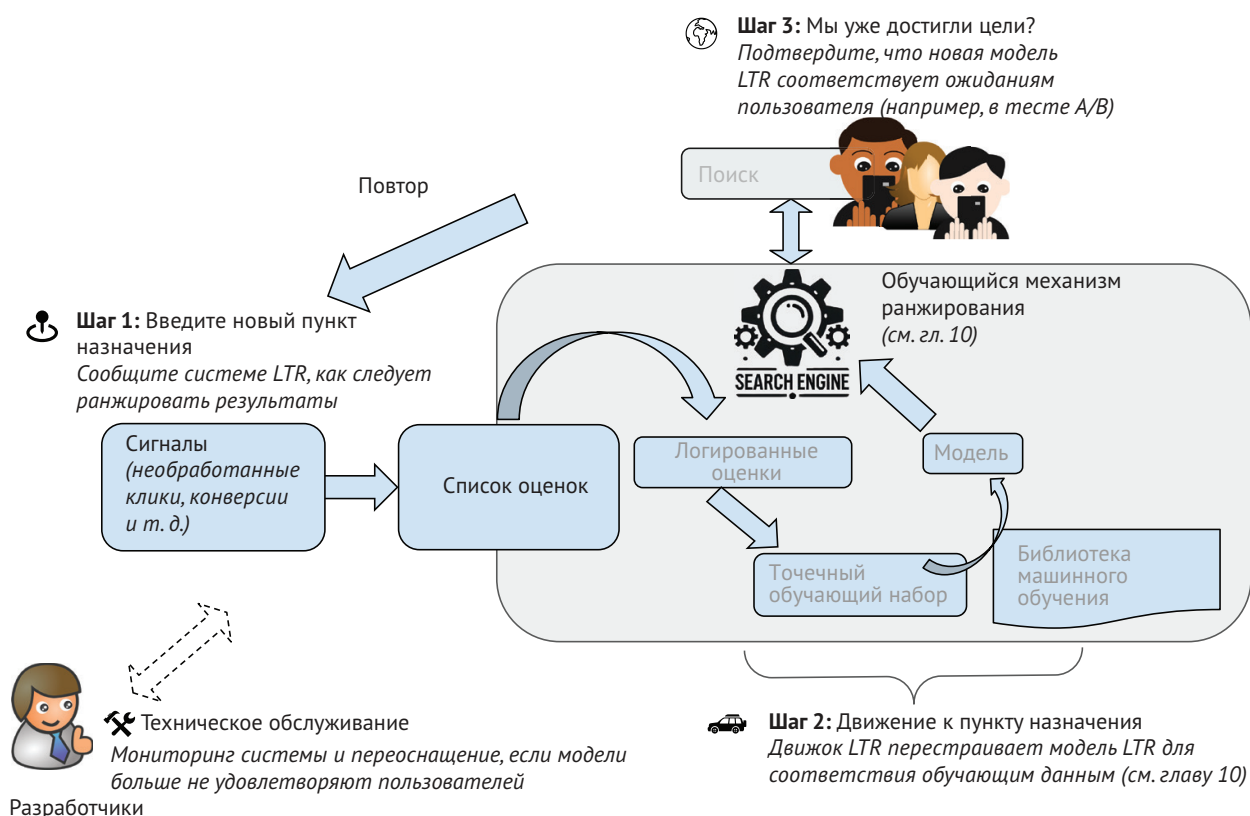


Рис. 11.1. Автоматизированная система LTR автоматически обучается и переобучается на основе сигналов пользователя. Это помогает строить модели на основе того, что фактические пользователи считают релевантным по многим запросам

Давайте кратко рассмотрим каждый шаг автоматизированного процесса LTR.

- 1 **Ввод нового пункта назначения** – мы вводим данные обучения в систему LTR, описывающие идеальную релевантность, на основе нашего

понимания поведенческих сигналов пользователей, таких как поиск, клики и конверсии (рассматривается в этой главе).

- 2 *Поездка к пункту назначения* – наша система LTR переобучает модель LTR, используя предоставленные данные обучения (рассматриваемые в главе 10).
- 3 *Мы уже достигли цели?* – действительно ли модель помогает пользователям? И возможно, будущие модели должны исследовать альтернативные маршруты (рассматриваются в главе 12)?

Автоматизированный LTR непрерывно повторяет шаги 1–3 для автоматической оптимизации релевантности. Поисковая группа отслеживает производительность автоматизированного LTR и вмешивается по мере необходимости. Это часть обслуживания¹ на рис. 11.1. Во время обслуживания мы открываем капот, чтобы исследовать новые признаки LTR и другие корректировки модели. Обслуживание также может означать повторный просмотр шага 1 для исправления нашего понимания поведения пользователей и создания более надежных данных обучения. В конце концов, без хороших обучающих данных мы могли бы следовать главе 10 до конца и все равно не удовлетворить наших пользователей.

В этой главе мы начинаем наше исследование автоматизированного LTR, сосредоточившись на шаге 1 – вводе нового пункта назначения. Сначала мы определим задачу получения обучающих данных из кликов пользователей. Затем посвятим оставшуюся часть этой главы преодолению некоторых распространенных предвзятостей и проблем с данными кликов поиска. К концу этой главы вы сможете строить модели с более надежными обучающими данными, полученными из сигналов пользователей. Затем глава 12 завершит наше автоматизированное исследование LTR, с наблюдением за взаимодействием модели с живыми пользователями и использованием активного обучения и гауссовых методов для преодоления более сложных предвзятостей представления и интегрируя все эти компоненты в окончательную сквозную автоматизированную систему LTR.

11.1. (Повторное) создание списков суждений из сигналов

Мы упоминали, что нам необходимо преодолевать предвзятости при создании обучающих данных LTR из кликов. Однако, прежде чем мы углубимся в эти предвзятости, мы рассмотрим последствия использования кликов вместо ручных меток для обучающих данных LTR. Затем мы сделаем наивную первую попытку создания обучающих данных

¹ Обслуживание, англ. *maintenance*, в программировании – это процесс модификации и обновления программного обеспечения после его развертывания. Его цель – исправить ошибки, улучшить производительность или другие признаки, а также адаптировать программное обеспечение к изменяющейся среде на протяжении всего срока его жизни. – Прим. ред.

в этом разделе, размышляя о том, что прошло хорошо или не очень. Это подготовит нас к остальной части главы, где мы рассмотрим устранение предвзятости из этих результатов (в разделе 11.2 и далее).

11.1.1. Генерация неявных вероятностных суждений из сигналов

Давайте зожим основу того, как использовать поведенческие сигналы в качестве обучающих данных LTR. Затем мы углубимся в детали построения надежных списков суждений.

В главе 10 мы обсуждали обучающие данные LTR, называемые *списками суждений* или *суждениями*. Эти суждения содержат метки или *оценки* (grades) того, насколько релевантны потенциальные результаты поиска для данного запроса. В главе 10 мы использовали фильмы в качестве примера, помечая их оценкой 1 (релевантно) или 0 (нерелевантно), как в следующем примере.

Листинг 11.1. Маркировка фильмов как релевантных или нерелевантных

```
# Judgment(grade, keywords, doc_id)
sample_judgments = [
    # for 'social network' query
    Judgment(1, "social network", 37799), # The Social Network
    Judgment(0, "social network", 267752), # #chicagoGirl
    Judgment(0, "social network", 38408), # Life As We Know It
    Judgment(0, "social network", 28303), # The Cheyenne Social Club
    # for 'star wars' query
    Judgment(1, "star wars", 11), # Star Wars
    Judgment(1, "star wars", 1892), # Return of the Jedi
    Judgment(0, "star wars", 54138), # Star Trek Into Darkness
    Judgment(0, "star wars", 85783), # The Star
    Judgment(0, "star wars", 325553) # Battlestar Galactica
]
```

Существует множество методов создания списков суждений, и эта глава не является исчерпывающей главой о списках суждений и их многочисленных приложениях. Вместо этого мы сосредоточимся на данных обучения LTR. По этой причине мы обсудим только суждения, сгенерированные из сигналов кликов пользователей. Мы называем их *неявными суждениями*, потому что они вытекают из взаимодействия пользователя с поисковым приложением, когда пользователи выполняют поиск и делают клики. Это контрастирует с *явными суждениями*, когда оценщики напрямую маркируют результаты поиска как релевантные или нерелевантные.

Неявные суждения идеально подходят для автоматизации LTR по нескольким причинам.

- Новизна – у нас есть готовый доступ к трафику пользователей, поэтому мы можем автоматизировать обучение сегодняшней модели LTR на основе последних ожиданий поиска пользователей.

- Больше данных с меньшими затратами – настройка задачи по сбору явных суждений (даже с краудсорсингом) требует много времени и затрат, чтобы сделать ее хорошо в масштабе. Получение неявных суждений из реальных взаимодействий пользователей, которые мы уже собираем, позволяет нам использовать существующую базу пользователей для выполнения этой работы за нас.
- Захват реальных случаев использования – неявные суждения захватывают реальных пользователей, выполняющих реальные задачи с вашим поисковым приложением. Сравните это с искусственной обстановкой, где явные оценщики тщательно решают, возможно, надуманно, абстрактную задачу выбора наиболее релевантных результатов.

К сожалению, данные о кликах могут содержать много шума. Мы не знаем, почему пользователь кликнул на данный результат поиска. Кроме того, пользователи неоднородны; некоторые будут интерпретировать один результат как релевантный, в то время как другие будут думать иначе. Поисковые взаимодействия содержат также предвзятости, которые необходимо преодолеть, создавая дополнительную неопределенность вокруг расчетов модели, которые мы подробно обсудим далее в этой и следующей главе.

По этим причинам вместо бинарного суждения модели кликов создают *вероятностные суждения*. Вместо того чтобы давать оценку только 1 (релевантно) или 0 (нерелевантно), оценка представляет собой вероятность того (между 0.0 и 1.0), что случайный пользователь посчитает результат релевантным или нет. Например, хорошая модель кликов может переформулировать суждения из листинга 11.1 как что-то более похожее на следующее.

Листинг 11.2. Маркировка релевантности запроса фильма вероятностным способом

```
# Judgment(grade, keywords, doc_id),
sample_judgments = [
    Judgment(0.99, "social network", 37799),      # The Social Network
    Judgment(0.01, "social network", 267752),     # #chicagoGirl
    Judgment(0.01, "social network", 38408),      # Life As We Know It
    Judgment(0.01, "social network", 28303),      # The Cheyenne Social Club
    Judgment(0.99, "star wars", 11),              # Star Wars
    Judgment(0.80, "star wars", 1892),            # Return of the Jedi
    Judgment(0.20, "star wars", 54138),           # Star Trek Into Darkness
    Judgment(0.01, "star wars", 85783),           # The Star
    Judgment(0.20, "star wars", 325553)           # Battlestar Galactica
]
```

Обратите внимание на фильмы «Звездные войны» в листинге 10.2 – оценка стала намного интереснее. Теперь у «Звездных войн» очень высокая вероятность релевантности (0.99). У сиквела «Возвращение джедая» вероятность немного ниже. Другие научно-фантастические фильмы («Звездный путь: Возмездие» и «Звездный крейсер «Галакти-

ка») имеют рейтинги немного выше 0, так как поклонникам франшизы «Звездные войны» также могут понравиться эти фильмы. «Звезда» с ними совершенно не связана – это детский анимационный фильм о первом Рождестве – поэтому она получает низкую вероятность релевантности 0.01.

11.1.2. Обучение модели LTR с использованием вероятностных суждений

Мы только что представили идею о том, что оценка релевантности может быть вероятностной. Теперь давайте рассмотрим, как мы можем применить уроки из главы 10 для обучения модели с использованием этих вероятностных суждений (от 0.0 до 1.0) вместо бинарных суждений.

Как правило, вы можете рассмотреть эти варианты при обучении модели:

- квантуйте оценки – проще говоря, вы можете установить произвольные пороговые значения перед обучением, чтобы преобразовать оценки в приемлемый формат. Вы можете назначить оценку выше 0.75 как релевантную (или 1.00). Все, что меньше 0.75, будет считаться нерелевантным (или 0.00). Другие алгоритмы, такие как LambdaMART, принимают диапазон оценок, например от 1 до 4, и они также могут иметь дискретные отсечки, например назначая всему, что меньше 0.25, оценку 1.00, всему, что больше или равно 0.25, но меньше 0.5, оценку 2.00 и т. д. С помощью этих алгоритмов вы можете перед обучением создать 100 таких меток, назначая 0.00 оценку 0, 0.01 оценку 1 и т. д., пока 1 не будет назначена оценка 100;
- Просто используйте суждения с плавающей точкой – алгоритм SVMRank из главы 10 вычитал признаки более релевантного предмета из признаков менее релевантного предмета (и наоборот) и построил классификатор, чтобы отличать релевантные предметы от нерелевантных. Мы сделали это с бинарными суждениями, но ничто не мешает нам делать это с вероятностными суждениями. Здесь, если «Return of the Jedi» (grade=0.80) считается более релевантным, чем «Star Trek Into Darkness» (grade=0.20), мы просто отмечаем «Return of the Jedi» как более релевантный, чем «Star Trek Into Darkness» (обозначая разницу как +1). Затем мы выполняем то же самое парное вычитание, которое мы бы выполнили из главы 10, вычитая признаки «Star Trek Into Darkness» из признаков «Return of the Jedi», чтобы создать полный пример обучения.

Переобучение модели с суждениями в этой главе в основном повторяет код из главы 10, поэтому вместо этого мы сосредоточимся на механике обучения модели кликов. Мы включили блокнот с полным сквозным примером обучения LTR (см. раздел 11.4), который интегрирует модель кликов, к которой мы придем к концу этой главы, в процесс обучения LTR, который вы уже изучили в главе 10.

Пора вернуться к коду и увидеть нашу первую модель кликов!

11.1.3. Показатель кликабельности: ваша первая модель кликов

Теперь, когда вы увидели формат суждений, который генерирует модель кликов, и как этот формат можно интегрировать для обучения модели LTR, давайте сделаем первый, наивный проход по построению модели кликов. После этого мы сделаем шаг назад, чтобы сосредоточиться на более сложной, универсальной модели кликов, а затем рассмотрим некоторые основные предвзятости, присущие обработке запросов и сигналов кликов.

СОВЕТ. Если вы хотите глубже погрузиться в эту тему, мы рекомендуем вам прочитать «Модели кликов для веб-поиска» Чуклина, Маркова и Рийке (Springer, 2015).

Чтобы построить нашу модель кликов, вернемся к набору данных RetroTech, поскольку он удобно связан с сигналами кликов пользователя. Из этих сигналов мы также провели обратную разработку типа необработанных данных сеанса, которые вам нужны для построения высококачественных суждений. Мы воспользуемся библиотекой `rapdas` для выполнения табличных вычислений на основе данных сеанса.

В следующем листинге мы рассмотрим пример сеанса поиска для фильма «Трансформеры: Темная сторона Луны». Эта необработанная информация о сеансе является отправной точкой – минимальной информацией, необходимой для разработки списка суждений на основе сигналов пользователя.

Листинг 11.3. Изучение сеанса поиска

```
query = "transformers dark of the moon"
sessions = get_sessions(query)
print(sessions.loc[3])
```

Выбирает сеансы для «transformers dark of the moon».

Проверяет один сеанс поиска, показанный пользователю.

Вывод:

sess_id	query	rank	doc_id	clicked
3	transformers dark of the moon	0.0	47875842328	False
3	transformers dark of the moon	1.0	24543701538	False
...				
3	transformers dark of the moon	7.0	97360810042	True
...				
3	transformers dark of the moon	13.0	47875841406	False
3	transformers dark of the moon	14.0	400192926087	False

Листинг 11.3 соответствует одному сеансу поиска с `sess_id=3` для запросов `transformers dark of the moon`. Этот сеанс включает запрос, ранжированные результаты, увиденные пользователем, и был ли кликнут каждый результат. Эти три предмета являются основными ингредиентами, необходимыми для построения модели кликов.

Сеансы поиска часто будут отличаться. Другой сеанс даже через несколько секунд может иметь немного другой рейтинг, представленный

пользователю. Индекс поиска мог измениться, или новый алгоритм релевантности мог быть развернут в производстве. Мы рекомендуем вам повторить листинг 11.3 с другим `sess_id` для сравнения сеансов.

Давайте преобразуем эти данные в суждения, используя нашу первую простую модель кликов: *показатель кликабельности*¹.

Создание суждений из показателя кликабельности

Мы начнем с построения очень простой модели кликов, чтобы освоиться с данными, а затем сможем сделать шаг назад, чтобы увидеть недостатки в этом первом проходе. Это позволит нам тщательно обдумать качество сгенерированных суждений для автоматизированного LTR в оставшейся части этой главы.

Наша первая модель кликов будет основана на показателе кликабельности (CTR). CTR – это количество кликов, полученных по результату поиска, деленное на количество раз, когда он появлялся в результатах поиска. Если результат кликается каждый раз, когда поисковая система возвращает результат, CTR будет равен 1. Если по нему никогда не кликают, CTR будет равен 0. Звучит достаточно просто – что может пойти не так?

Мы можем просмотреть каждый результат для запроса `transformers dark of the moon` и рассмотреть клики относительно количества сеансов, в которых был возвращен `doc_id`. Следующий листинг показывает вычисление и результирующее значение CTR для каждого документа.

Листинг 11.4. Вычисление CTR

```
def calculate_ctr(sessions):
    click_counts = sessions.groupby("doc_id")["clicked"].sum()
    sess_counts = sessions.groupby("doc_id")["sess_id"].nunique()
    ctrs = click_counts / sess_counts
    return ctrs.sort_values(ascending=False)

query = "transformers dark of the moon"
sessions = get_sessions(query, index=False)
click_through_rates = calculate_ctr(sessions)
print_series_data(click_through_rates, column="CTR")
```

Вывод:

doc_id	CTR	name
97360810042	0.0824	Transformers: Dark of the Moon - Blu-ray Disc
47875842328	0.0734	Transformers: Dark of the Moon Stealth Force
E...		

¹ Показатель кликабельности, англ. *Click-Through Rate (CTR)*, – это процентное соотношение тех, кто увидел рекламу, к тем, кто кликнул на объявление. Чем выше процент, тем интереснее реклама пользователям. CTR помогает оценить привлекательность и релевантность контента. На него ориентируются, когда нужно привести потенциальных клиентов и получить заявки или продажи. – Прим. ред.

```

47875841420    0.0434    Transformers: Dark of the Moon Decepticons - ...
...
93624956037    0.0082    Transformers: Dark of the Moon - Original
Soun...
47875841369    0.0074    Transformers: Dark of the Moon - PlayStation 3
24543750949    0.0062    X-Men: First Class - Widescreen Dubbed
Subtitl...

```

В листинге 11.4 для всех сеансов с запросом `transformers dark of the moon` (согласно листингу 11.3) мы суммируем клики для каждого `doc_id` как `click_counts`. Мы также подсчитываем количество уникальных сеансов для этого документа в `sess_counts`. Наконец, мы вычисляем `ctr` как `click_counts / sess_counts`, что дает нам нашу первую модель кликов. Мы видим, что документ 97360810042 имеет самый высокий CTR, а 24543750949 – самый низкий.

Предыдущий листинг выводит идеальные результаты поиска на основе CTR. То есть, если бы наша модель LTR была обучена с использованием этой модели кликов CTR для предоставления суждений о релевантности, поисковая система выдала бы этот порядок как оптимальный рейтинг. В этой и следующей главах мы часто будем визуально отображать этот идеальный рейтинг, чтобы понять, создает ли модель кликов разумные данные для обучения (суждения). Мы можем увидеть идеальные суждения на основе CTR для `transformers dark of the moon` на рис. 11.2.

Оценки CTR для `q=transformers dark of the moon`

	ctr	upc	image	name
0	0.0824	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1	0.0734	47875842328		Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii
2	0.0434	47875841420		Transformers: Dark of the Moon Decepticons - Nintendo DS
3	0.0364	24543701538		The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc
4	0.0352	25192107191		Fast Five - Widescreen - Blu-ray Disc

Рис. 11.2. Результаты поиска, ранжированные по CTR для запроса `transformers dark of the moon`

При рассмотрении результатов рис. 11.2 бросается в глаза пара вещей:

- CTR для нашего главного результата (Blu-ray с фильмом «Transformers: Dark of the Moon») кажется довольно низким (0.0824, лишь немного лучше, чем следующее суждение 0.0734). Мы могли бы ожидать, что степень релевантности Blu-ray будет намного выше, чем у других результатов;
- DVD с фильмом «Transformers: Dark of the Moon» даже не отображается. Он находится намного ниже, казалось бы, не связанных между собой фильмов и второстепенных видеоигр о фильме «Dark of the Moon». Мы ожидали бы, что DVD будет ранжироваться выше, может быть, так же высоко или выше, чем Blu-ray.

Но, возможно, transformers dark of the moon – это просто странный запрос. Давайте повторим процесс для чего-то совершенно не связанного, на этот раз для сушилки на рис. 11.3.

CTR для q=dryer

	ctr	upc	image	name
0	0.1608	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
1	0.0816	84691226703		Hotpoint - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White-on-White
2	0.0710	12505451713		Frigidaire - Semi-Rigid Dryer Vent Kit - Silver
3	0.0576	783722274422		The Independent - Widescreen Subtitle - DVD
4	0.0572	883049066905		Whirlpool - Affresh Washer Cleaner

Рис. 11.3. Результаты поиска, ранжированные по CTR для запроса dryer. Здесь мы отмечаем странный результат для фильма «The Independent», который не кажется релевантным

На рис. 11.3 мы видим другие странно выглядящие результаты:

- первые два результата – сушилки для белья, что кажется хорошим;
- после сушилок для белья идут детали сушилок для белья. Хм, ладно?
- появляется фильм под названием «The Independent». Это кажется совершенно случайным. Почему он получил такую высокую оценку?

- далее идет аксессуар для стиральной машины, который в некотором роде связан с темой запроса;
- наконец, мы видим фены, что показывает еще одно потенциальное значение слова «сушилка».

Что вы думаете о суждениях, выдаваемых моделью кликов CTR? Вспомните, что вы узнали в главе 10. Помните, что это основа, сама цель вашей модели LTR. Как вы думаете, приведут ли эти суждения к хорошей модели LTR, которая в конечном итоге будет успешной, если ее внедрить в производство?

Мы также призываем вас задать себе более фундаментальный вопрос: как мы вообще можем определить, хорош ли список суждений? Наша субъективная интерпретация может быть столь же несовершенной, как и данные в модели кликов. Мы рассмотрим это более аналитически в главе 12. В этой главе мы позволим нашим инстинктам направить нас к возможным проблемам.

11.1.4. Распространенные предвзятости в суждениях

До сих пор мы видели, что можем создавать вероятностные суждения – с оценками от 0.00 до 1.00, – просто разделив количество кликов по продукту на количество раз, когда этот продукт был возвращен поиском. Однако вывод, похоже, был немного недостаточным, поскольку он включал фильмы, не связанные с франшизой «Transformers». Мы также видели фильм, размещенный в результатах поиска по запросу *dyer*!

Оказывается, данные о кликах поиска полны предвзятостей (предвзятостей). Здесь мы кратко определим, что мы подразумеваем под предвзятостью, прежде чем исследовать каждую из этих предвзятостей в данных о кликах RetroTech.

В моделях кликов *предвзятость* является причиной того, что необработанные данные о кликах пользователя могут не иметь ничего общего с релевантностью результатов поиска. Вместо этого предвзятости определяют, как клики (или отсутствие кликов) отражают психологию пользователя, дизайн пользовательского интерфейса поиска или зашумленные данные. Мы можем разделить предвзятости на две большие группы: неалгоритмические и алгоритмические предвзятости. *Алгоритмические предвзятости* присущи ранжированию, отображению и взаимодействию с результатами поиска. *Неалгоритмические предвзятости* возникают по причинам, только косвенно связанным с ранжированием поиска.

Алгоритмические предвзятости могут включать следующее:

- предвзятость позиции – пользователи кликают на результаты с более высоким рейтингом чаще, чем на результаты с более низким рейтингом;
- предвзятость уверенности – документы с небольшим количеством сигнальных данных влияют на суждения так же, как и документы с гораздо большим количеством данных;
- предвзятость показа – если поиск никогда не показывает определенные результаты, пользователи никогда не кликают на них, поэтому модель кликов не будет знать, являются ли они релевантными.

С другой стороны, неалгоритмические предвзятости – это предвзятости, подобные следующим:

- предвзятость привлекательности – некоторые результаты кажутся привлекательными и генерируют клики (возможно, из-за лучших изображений или выбора формулировок), но они оказываются спамом или просто нерелевантными;
- предвзятость производительности – пользователи отказываются от медленного поиска, отвлекаются и в конечном итоге ничего не кликают или кликают только на самые ранние возвращенные результаты.

Поскольку эта книга о *поиске на основе ИИ*, мы сосредоточимся на обсуждении *алгоритмических* предвзятостей в данных о кликстриме¹ поиска. В этой главе мы рассмотрим предвзятость позиции и предвзятость уверенности. Предвзятость презентации будет рассмотрена в главе 12.

Но неалгоритмические предвзятости тоже имеют значение! Поиск – это сложная экосистема, которая выходит за рамки рейтингов релевантности. Если результаты часто кликаются, но последующие действия, такие как продажи или другие конверсии, не происходят, это может быть не проблема ранжирования – возможно, у вас проблема со спам-продуктами. Или у вас может быть проблема со страницами продуктов или процессом оформления заказа. Вас могут попросить улучшить релевантность, когда ограничивающим фактором на самом деле является пользовательский опыт, контент или скорость поиска.

Теперь, когда мы размышляли о нашей первой модели клика, давайте поработаем над преодолением первой предвзятости.

11.2. Преодоление предвзятости позиции

В предыдущем разделе мы увидели нашу первую модель кликов в действии: простую модель кликов CTR. Она делила количество кликов по продукту в поиске на количество его возвратов в верхних результатах. Мы увидели, что это был довольно ошибочный подход, отметив многочисленные причины, по которым он мог быть предвзятым. В частности, мы указали на предвзятость позиции, предвзятость уверенности и предвзятость презентации как на три алгоритмических предвзятости, присутствующих в нашей модели кликов. Пришло время начать решать эти проблемы!

В этом разделе мы сосредоточимся на первой из этих алгоритмических предвзятостей, предвзятости позиции, углубимся в проблему и поработаем над моделью кликов, предназначенной для ее преодоления.

¹ Кликстрим, англ. *clickstream*, – это запись переходов по страницам, кликов по изображениям, выставление оценок и т. д.. Анализ таких данных позволяет получить представление о поведении пользователей, определить популярный контент, оптимизировать навигацию на сайте и улучшить общий пользовательский опыт. – *Прим. ред.*

11.2.1. Определение предвзятости позиции

Предвзятость позиции присутствует в большинстве поисковых систем. Если пользователям показываются результаты поиска, они, как правило, предпочитают высокоранжированные результаты поиска более низким, даже если эти более низкие результаты на самом деле более релевантны. Йоахимс и др. В своей статье «Оценка точности неявной обратной связи от кликов и переформулировок запросов в веб-поиске» (www.cs.cornell.edu/people/tj/publications/joachims_etal_07a.pdf) обсуждают несколько причин существования предвзятости позиции.

- Предвзятость достоверности – пользователи верят, что поисковая система должна знать, что она делает, поэтому они больше взаимодействуют с более высокими результатами.
- Манера сканирования – пользователи изучают результаты поиска по определенным шаблонам, например перескакивают сверху вниз, и часто не изучают все, что находится перед ними.
- Видимость – результаты с более высоким рейтингом, скорее всего, будут отображаться на экране пользователя, поэтому пользователям пришлось бы прокрутить страницу, чтобы увидеть оставшиеся результаты.

С учетом этих факторов давайте посмотрим, сможем ли мы обнаружить предвзятость позиции в сеансах RetroTech.

11.2.2. Предвзятость позиции в данных RetroTech

Какова степень предвзятости позиции в сеансах в наборе данных RetroTech? Если мы можем количественно оценить это, мы можем рассмотреть, как именно можно исправить эту проблему. Давайте быстро оценим предвзятость, прежде чем рассматривать новую модель кликов для преодоления этих предвзятостей.

Просматривая все сеансы по всем запросам, мы можем вычислить средний CTR на ранг. Это скажет нам, насколько сильно предвзятость позиции проявляется в данных о кликах RetroTech. Мы делаем это в следующем листинге.

Листинг 11.5. CTR на ранг в сеансах поиска по всем запросам

```
sessions = all_sessions()
num_sessions = len(sessions["sess_id"].unique())
ctr_by_rank = sessions.groupby("rank")["clicked"].sum() / num_sessions
print(ctr_by_rank)
```

Вывод:

```
rank
0      0.249727
1      0.142673
2      0.084218
```

3	0.063073
4	0.056255
5	0.042255
6	0.033236
7	0.038000
8	0.020964
9	0.017364
10	0.013982

Вы можете видеть в листинге 11.5, что пользователи чаще кликают на более высокие позиции. CTR результатов с рангом 0 составляет 0.25, затем следует 0.143 с рангом 1 и т. д.

Кроме того, мы можем увидеть предвзятость позиции, когда сравниваем суждения о CTR более раннего периода с типичным рейтингом для каждого продукта в запросе. Если предвзятость позиции присутствует, то идеальный рейтинг нашего суждения в конечном итоге будет напоминать типичный рейтинг, показываемый пользователям. Мы можем проанализировать это, усреднив ранг каждого документа за каждый сеанс, чтобы увидеть, где они появляются.

В следующем листинге показана типичная страница результатов поиска для сеансов transformers dark of the moon.

Листинг 11.6. Изучение ранжирования для transformers dark of the moon

```
def calculate_average_rank(sessions):
    avg_rank = sessions.groupby("doc_id")["rank"].mean()
    return avg_rank.sort_values(ascending=True)

sessions = get_sessions("transformers dark of the moon")
average_rank = calculate_average_rank(sessions)
print_series_data(average_rank, "mean_rank")
```

Вывод:

doc_id	mean_rank	name
400192926087	13.0526	Transformers: Dark of the Moon - Original Soun...
97363532149	12.1494	Transformers: Revenge of the Fallen - Widescre...
93624956037	11.3298	Transformers: Dark of the Moon - Original Soun...
...		
25192107191	2.6596	Fast Five - Widescreen - Blu-ray Disc
24543701538	1.8626	The A-Team - Widescreen Dubbed Subtitle AC3 - ...
47875842328	0.9808	Transformers: Dark of the Moon Stealth Force E...

В листинге 11.6 некоторые документы, такие как 24543701538 и 47875842328, исторически появляются в верхней части результатов поиска для этого запроса. Они будут чаще кликаться из-за предвзятости позиции. Типичная страница результатов, показанная на рис. 11.4, довольно сильно пересекается с рейтингом CTR из рис. 11.2.

Типичный сеанс поиска для q=transformers dark of the moon.

	mean_rank	upc	image	name
0	0.9808	47875842328		Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii
1	1.8626	24543701538		The A-Team - Widescreen Dubbed Subtitle AC3 - Blu-ray Disc
2	2.6596	25192107191		Fast Five - Widescreen - Blu-ray Disc
3	3.5344	47875841420		Transformers: Dark of the Moon Decepticons - Nintendo DS
4	4.4444	786936817218		Pirates of the Caribbean: On Stranger Tides - Blu-ray 3D

Рис. 11.4. Типичная страница результатов поиска для запроса transformers dark of the moon. Обратите внимание на нерелевантные фильмы, такие как «The A-Team» и «Fast Five». Также обратите внимание на высокий рейтинг игры Wii. Высокая позиция этих результатов и тот факт, что они получают больше кликов, просто показываясь выше в списке, объясняет, почему модель CTR ошибочно считает их релевантными

К сожалению, на CTR в первую очередь влияет предвзятость позиции. Пользователи кликают на странные фильмы на рис. 11.4, потому что поисковая система возвращает их высоко для этого запроса, а не потому, что они релевантны. Если мы обучаем модель LTR только на CTR, мы попросим модель LTR оптимизироваться в плане того, что пользователи уже видят и с чем взаимодействуют. Мы должны учитывать предвзятость позиции при автоматизации LTR.

Далее, давайте посмотрим, как мы можем преодолеть предвзятость позиции в более надежной модели кликов, которая компенсирует предвзятость позиции.

11.2.3. Упрощенная динамическая байесовская сеть: модель кликов, которая преодолевает предвзятость позиции

Вы увидели вред, который предвзятость позиции может нанести в действии! Если мы просто будем использовать клики напрямую, мы обучим нашу модель LTR, чтобы усилить рейтинг, уже показанный пользователям. Пришло время представить модель кликов, которая может преодолеть предвзятость позиции. Начнем с определения «examine» (изучение), ключевого понятия в моделировании предвзятости пози-

ции. Затем мы представим одну конкретную модель кликов, которая использует это понятие для корректировки необработанных кликов с целью преодоления предвзятости позиции.

Как модели клик-кода преодолевают предвзятость позиции с помощью события «examine»

Базовый расчет CTR на самом деле не учитывает, как пользователи просматривают результаты поиска. Пользователь, скорее всего, рассматривает только несколько результатов – смещенных по положению, – прежде чем решить кликнуть один или два. Если мы сможем определить, какие результаты пользователи сознательно рассматривают перед кликом, то сможем преодолеть предвзятость позиции. Модели кликов делают именно это, определяя понятие *изучения*. Мы рассмотрим это понятие, прежде чем строить модель клика, которая преодолевает предвзятость позиции.

Что такое изучение? Вы, возможно, знакомы с *представлением* (impression) – когда элемент пользовательского интерфейса отображается на видимой части экрана пользователя. В моделях кликов мы вместо этого рассматриваем *изучение*, вероятность того, что результат поиска был рассмотрен пользователем сознательно. Как мы знаем, пользователи часто не замечают что-то, находящееся прямо перед их глазами. Возможно, вы даже были этим пользователем! Рисунок 11.5 отражает это понятие, противопоставляя представление и изучение.

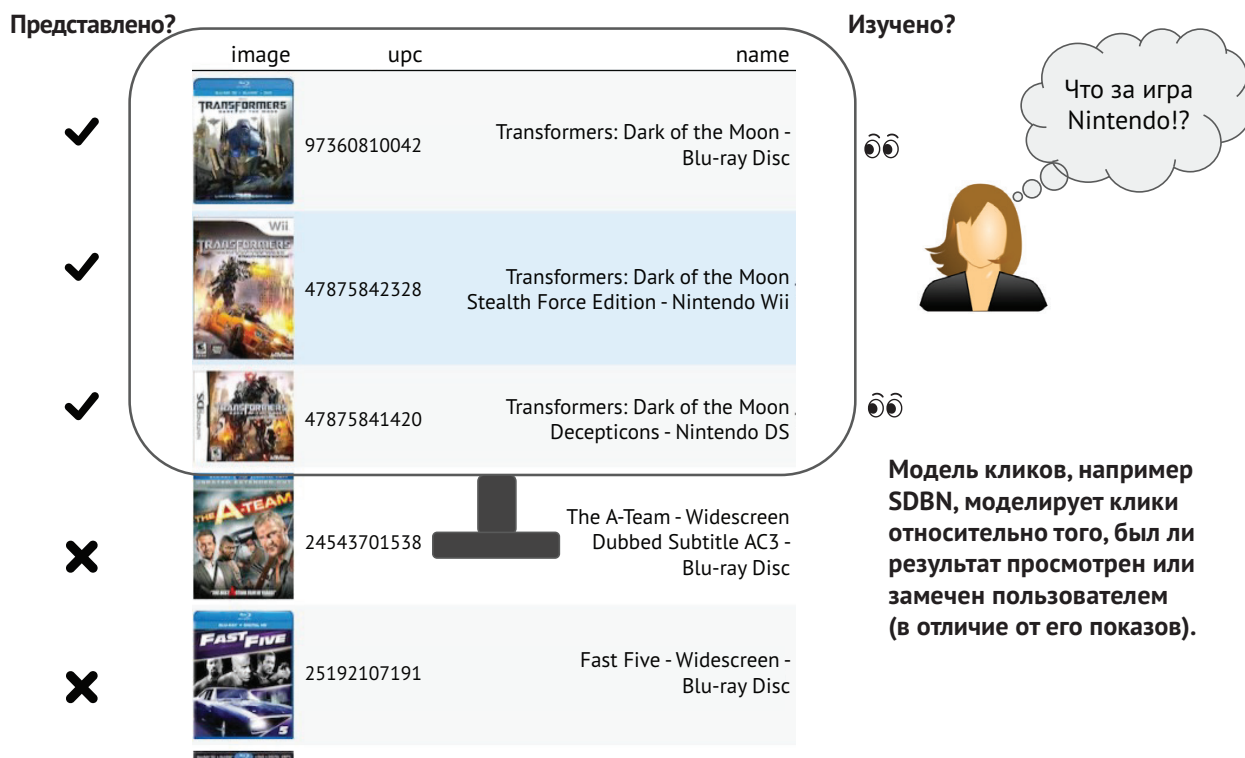


Рис. 11.5. Представление – это то, что отображается в области просмотра (квадрат в форме монитора), в то время как изучение – это то, что пользователь рассматривает (результаты с соседними глазными яблоками). Моделирование того, что пользователи изучают, помогает правильно учитывать, как пользователи взаимодействуют с результатами поиска

На рис. 11.5 вы можете видеть, что пользователь не замечает игру Nintendo во второй позиции, даже если она отображается на его мониторе. Если пользователь не заметил ее, модель кликов наказывать за релевантность игры Nintendo не должна.

Почему отслеживающие изучения помогают преодолеть предвзятость позиции? Изучения – это то, как модель кликов понимает предвзятость позиции. Другой способ сказать «предвзятость позиции» – «мы считаем, что то, изучают ли пользователи результаты поиска, зависит от позиции». В результате моделирование правильного изучения является основной деятельностью большинства моделей кликов. Некоторые модели кликов, такие как *модель на основе позиции* (PBM), пытаются определить вероятность изучения на каждую позицию по всем поискам. Другие, такие как *каскадная модель* или, как мы скоро увидим, *модель динамических байесовских сетей* (DBN), предполагают, что, если результат был выше последнего клика на странице поиска, он, скорее всего, был изучен.

Для большинства моделей кликов верхняя позиция обычно имеет более высокую вероятность изучения, чем нижние. Это позволяет моделям кликов правильно подстраиваться под клики. Предметы, которые часто изучаются и кликаются, вознаграждаются и рассматриваются как более релевантные. Те, которые изучаются, но не кликаются, рассматриваются как менее релевантные.

Чтобы сделать это более конкретным, давайте углубимся в одну из моделей динамической байесовской сети кликов, которая использует изучения для преодоления предвзятости позиции.

Определение упрощенной динамической байесовской сети

Упрощенная динамическая байесовская сеть (SDBN) является немного менее точной версией более сложной модели динамической байесовской сети кликов (DBN). Эти модели кликов предполагают, что в рамках сеанса поиска вероятность того, что пользователь изучил документ, сильно зависит от того, был ли он расположен на уровне или выше самого низкого клика. Алгоритм SDBN сначала отмечает последний клик каждого сеанса, а затем считает каждый документ на уровне или выше этого последнего клика как проверенный. Наконец, он вычисляет степень релевантности, просто разделив общее количество кликов по документу на общее количество просмотров этого документа. Таким образом, мы получаем своего рода динамический CTR, отслеживая в рамках сеанса поиска каждого пользователя, когда он, вероятно, изучил результат, и тщательно используя это для учета того, как этот пользователь оценил его релевантность. Затем мы используем эти оценки релевантности в совокупности по сеансам для обучения модели кликов SDBN.

Давайте следовать этому алгоритму шаг за шагом. Сначала мы отметим последний клик каждого сеанса в следующем листинге.

Листинг 11.7. Отметка того, какие результаты
были изучены в каждом сеансе

```
def calculate_examine_probability(sessions):
    last_click_per_session = sessions.groupby(
        ["clicked", "sess_id"])["rank"].max()[True]
    sessions["last_click_rank"] = last_click_per_session
    sessions["examined"] = \
        sessions["rank"] <= sessions["last_click_rank"]
    return sessions

sessions = get_sessions("dryer")
probability_data = calculate_examine_probability(sessions).loc[3]
print(probability_data)
```

Вычисляет last_click_per_session, максимальный рейтинг, где clicked равен True за сеанс.

Отмечает рейтинг последнего клика в каждом сеансе.

Устанавливает каждую позицию на уровне или выше последнего клика как True (иначе False)

Вывод (усеченный):

sess_id	query	rank	doc_id	clicked	last_click_rank	examined
3	dryer	0.0	12505451713	False	9.0	True
3	dryer	1.0	84691226727	False	9.0	True
3	dryer	2.0	883049066905	False	9.0	True
...						
3	dryer	8.0	14381196320	True	9.0	True
3	dryer	9.0	74108096487	True	9.0	True
3	dryer	10.0	74108007469	False	9.0	False
3	dryer	11.0	12505525766	False	9.0	False
...						

В листинге 11.7 мы находим максимальный ранг, где clicked равен True, сохраняя его в last_click_per_session. Затем отмечаем позиции на уровне или выше last_click_rank как проверенные в наших сеансах для dryer, как вы можете видеть в выводе для sess_id=3.

С каждым сеансом, обновленным с изучениями, установленным на True или False, мы теперь суммируем общее количество кликов и изучений на документ по всем сеансам.

Листинг 11.8. Сумма кликов и изучений на doc_id для этого запроса

```
def calculate_clicked_examined(sessions):
    sessions = calculate_examine_probability(sessions)
    return sessions[sessions["examined"]] \
        .groupby("doc_id")[["clicked", "examined"]].sum()

sessions = get_sessions("dryer")
clicked_examined_data = calculate_clicked_examined(sessions)
print_dataframe(clicked_examined_data)
```

Вывод (усеченный):

doc_id	clicked	examined	name
12505451713	355	2707	Frigidaire - Semi-Rigid Dryer Ve...
12505525766	268	974	Smart Choice - 6' 30 Amp 3-Prong...
...			
36172950027	97	971	Tools in the Dryer: A Rarities C...
...			
883049066905	286	2138	Whirlpool - Affresh Washer Cleaner
883929085118	44	578	A Charlie Brown Christmas - AC3 ...

В листинге 11.8 `sessions[sessions["examined"]]` фильтрует только изученные строки. Затем для каждого `doc_id` мы вычисляем общее количество кликов и изучений. Вы можете видеть, что некоторые результаты, такие как `doc_id=36172950027`, явно были изучены много с относительно небольшим количеством кликов от пользователей.

Наконец, мы завершаем алгоритм SDBN в следующем листинге, вычисляя клики по изучениям.

Листинг 11.9. Вычисляем окончательные оценки SDBN

```
def calculate_grade(sessions):
    sessions = calculate_clicked_examined(sessions)
    sessions["grade"] = sessions["clicked"] / sessions["examined"]
    return sessions.sort_values("grade", ascending=False)

query = "dryer"
sessions = get_sessions(query)
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

Вывод (усеченный):

doc_id	clicked	examined	grade	name
856751002097	133	323	0.411765	Practecol - Dryer Balls (2-Pack)
48231011396	166	423	0.392435	LG - 3.5 Cu. Ft. 7-Cycle High-Ef...
84691226727	804	2541	0.316411	GE - 6.0 Cu. Ft. 3-Cycle Electri..
...				
12505451713	355	2707	0.131141	Frigidaire - Semi-Rigid Dryer Ve..
36172950027	97	971	0.099897	Tools in the Dryer: A Rarities C..
883929085118	44	578	0.076125	A Charlie Brown Christmas - AC3
...				

В выводе листинга 11.9 документ 856751002097 рассматривается как наиболее релевантный с оценкой 0.4118, или 133 клика из 323 изучений.

Давайте вернемся к нашим двум запросам, чтобы увидеть, как теперь выглядят идеальные результаты для `dryer` и `transformers dark of the moon`. На рис. 11.6 показаны результаты для `dryer`, а на рис. 11.7 показаны результаты `transformers dark of the moon`.

Оценки SDBN для q=dryer

grade	upc	image	name
0 0.4118	856751002097		Practecol - Dryer Balls (2-Pack)
1 0.3924	48231011396		LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White
2 0.3164	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
3 0.2938	74108007469		Conair - 1875-Watt Folding Handle Hair Dryer - Blue
4 0.2752	12505525766		Smart Choice - 6' 30 Amp 3-Prong Dryer Cord

Рис. 11.6. Идеальные результаты поиска для запроса dryer согласно SDBN. Обратите внимание, как SDBN, похоже, продвигает больше результатов, связанных со стиркой одежды

Оценки SDBN для q=transformers dark of the moon

grade	upc	image	name
0 0.6417	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1 0.4806	400192926087		Transformers: Dark of the Moon - Original Soundtrack - CD
2 0.3951	97363560449		Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD
3 0.3231	97363532149		Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD
4 0.2662	93624956037		Transformers: Dark of the Moon - Original Soundtrack - CD

Рис. 11.7. Идеальные результаты поиска для запроса transformers dark of the moon, по версии SDBN. Теперь мы обнаружили DVD, Blu-ray-фильм и CD-саундтрек

Если мы субъективно рассмотрим рис. 11.6 и 11.7, оба набора суждений кажутся более интуитивными, чем предыдущие суждения STR. В нашем примере с сушилкой акцент, по-видимому, делается на стирке одежды. Есть некоторые аксессуары (например, шарики для сушилки), которые оцениваются примерно так же, как и сами сушилки.

Для transformers dark of the moon мы отмечаем очень высокую оценку Blu-ray. Мы также видим, что DVD и CD-саундтрек имеют более высокий рейтинг, чем другие вторичные предметы «Dark of the Moon», такие как видеоигры. Несколько странно, что CD с саундтреком имеет более высокий рейтинг, чем DVD с фильмом – возможно, нам следует изучить это подробнее.

Конечно, как мы уже говорили ранее, пока мы используем нашу интуицию. В главе 12 мы более объективно подумаем о том, как можно оценить качество суждений.

После того как наша предвзятость позиции стала более контролируемой, мы теперь перейдем к тонкой настройке наших суждений, чтобы справиться с другой важной предвзятостью: предвзятостью уверенности.

11.3. Управление предвзятостью уверенности: не пересматривать свою модель из-за нескольких удачных кликов

При игре в бейсбол средний показатель отбивания игрока говорит нам о доле попаданий по мячу при отбивании. У великого профессионального игрока средний показатель отбивания больше 0.3. Однако рассмотрим удачливого игрока младшей бейсбольной лиги, который выходит на позицию для своего первого отбивания и попадает по мячу. Его средний показатель отбивания технически равен 1.0! Таким образом, мы можем заключить, что этот маленький ребенок – вундеркинд в бейсболе и, безусловно, сделает отличную карьеру в бейсболе. Верно?

Не совсем так! В этом разделе мы рассмотрим релевантную сторону этого счастливого ребенка младшей лиги. Что нам делать с результатами, которые, возможно, просто по невезению были проверены всего несколько раз и каждый раз приводили к клику? Они, вероятно, не должны получить идеальную оценку 1.0. Мы увидим (и исправим) эту проблему в наших данных.

11.3.1. Проблема низкой достоверности в данных о кликах

Давайте посмотрим на данные, чтобы увидеть, где точки данных с низкой достоверностью портят обучающие данные. Затем посмотрим, как мы можем компенсировать проблемы низкой достоверности в результатах SDBN. Чтобы определить проблему, давайте посмотрим на результаты SDBN для transformers dark of the moon и другой, более редкий запрос, чтобы увидеть распространенные ситуации низкой достоверности.

Если вы помните, было немного подозрительно, что саундтрек к фильму «Transformers Dark of The Moon» получил такой высокий рейтинг, по версии SDBN. Когда мы изучим необработанные данные, лежащие в основе рейтингов, мы сможем увидеть возможную проблему. В следующем листинге мы реконструируем данные SDBN для transformers dark of the moon, чтобы устранить эту проблему, объединяя листинги 11.7–11.9.

Листинг 11.10. Пересчет статистики SDB

```
query = "transformers dark of the moon"
sessions = get_sessions(query)
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

Вывод (усеченный):

doc_id	clicked	examined	grade	name
97360810042	412	642	0.641745	Transformers: Dark of the Moon -...
400192926087	62	129	0.480620	Transformers: Dark of the Moon -...
97363560449	96	243	0.395062	Transformers: Dark of the Moon -...
...				
47875841406	80	626	0.127796	Transformers: Dark of the Moon A...
24543750949	31	313	0.099042	X-Men: First Cla-s - Widescreen ...
47875842335	53	681	0.077827	Transformers: Dark of the Moon S...

В выводе листинга 11.10 обратите внимание, что верхний результат, фильм Blu-ray (doc_id=97360810042), имеет гораздо больше изучений (642), чем CD с саундтреком (doc_id=400192926087 с 129 изучениями). Оценка Blu-ray более надежна, учитывая, что у него было во много раз больше возможностей для взаимодействия с пользователем, что делает его менее подверженным доминированию шумных, ложных кликов. С другой стороны, у CD гораздо меньше изучений. Не следует ли придать большее значение оценке релевантности Blu-ray, учитывая, что это более надежная точка данных по сравнению с CD с более ограниченными данными взаимодействия с пользователем?

Часто эта ситуация еще более очевидна, особенно при работе с менее распространенными запросами. Независимо от количества запросов, которые получает ваша поисковая система, некоторые запросы, скорее всего, будут получены много раз (*головные запросы*), некоторые – умеренное количество раз (*запросы тела*), а некоторые – очень редко (*запросы длинного хвоста*, или просто *запросы хвоста*).

Рассмотрим запрос blue ray. Вы заметите, что это распространенная ошибка в написании «Blu-ray». Распространенной ошибкой является смешивание документов с небольшим количеством изучений с документами, которые выводятся редко. В следующем листинге мы вычисляем статистику SDBN для blue ray, которая страдает от этой проблемы разреженности данных.

Листинг 11.11. Оценки SDBN для запроса с разреженными данными

```
def get_sample_sessions(query):
    sessions = get_sessions(query, index=False)
    sessions = sessions[sessions["sess_id"] < 50050]
    return sessions.set_index("sess_id")

sessions = get_sample_sessions("blue ray")
grade_data = calculate_grade(sessions)
print_dataframe(grade_data)
```

Случайно выбирает несколько сеансов для имитации типичного случая длинного хвоста.

Вывод (усеченный):

doc_id	clicked	examined	grade	name
600603132872	1	1	1.000000	Blu-ray Disc Cases (10-Pack)
827396513927	14	34	0.411765	Panasonic - Blu-Ray Player
25192073007	8	20	0.400000	The Blues Brothers - Widescreen...
...				
25192107191	0	7	0.000000	Fast Five - Widescreen - Blu-r...
23942972389	0	15	0.000000	Verbatim - 10-Pack 6x BD-R Dis...
885170038875	0	5	0.000000	Panasonic - 9" Widescreen Port...

Глядя на вывод листинга 11.11, мы видим нечто тревожное. Как и в самом экстремальном случае с нашим счастливым игроком младшей лиги бейсбола, наиболее релевантный результат, doc 600603132872, получает оценку 1.0 (совершенно релевантный) после того, как его проверил всего один пользователь! Эта оценка 1.0 ранжируется выше, чем следующий результат, который имеет оценку 0.411 на основе 34 изучений. Если учесть, что doc 600603132872 – это набор случаев Blu-ray, а 827396513927 – это проигрыватель Blu-ray, это кажется более тревожным. Наша субъективная интерпретация может поставить игрока выше случаев. Разве тот факт, что второй результат был проверен значительно больше, не должен что-то значить?

То, что мы увидели в этих примерах, – это *предвзятость уверенности* – когда список суждений имеет много оценок, основанных на статистически незначимых, ложных событиях. Мы говорим, что эти ложные события с небольшим количеством изучений имеют низкую уверенность, тогда как те, у которых изучений больше, обеспечивают более высокий уровень уверенности. Независимо от вашей модели кликов, у вас, вероятно, будет много ситуаций, когда запросы имеют лишь скромный объем трафика. Чтобы автоматизировать LTR, вам нужно будет настроить генерацию обучающих данных, чтобы учесть вашу уверенность в данных.

Теперь, когда вы увидели эффект данных с низкой уверенностью, мы можем перейти к некоторым решениям, которые вы можете применить при построении своей модели кликов.

11.3.2. Использование бета-приорного распределения для моделирования достоверности вероятностным образом

Мы только что рассмотрели несколько проблем, возникающих из-за слишком высокой оценки данных с низкой достоверностью. Без настройки ваших моделей на основе вашей уверенности в данных вы не сможете построить надежную автоматизированную систему LTR. Мы могли бы просто отфильтровать эти примеры с низкой достоверностью, но, возможно, мы можем сделать что-то умнее? Мы обсудим подход к сохранению всех данных о потоке кликов в этом разделе, когда представим понятие бета-распределений. Но сначала давайте обсудим, почему использование всех данных, как правило, предпочтительнее, чем простое отфильтровывание примеров с низкой достоверностью.

Нужно ли отфильтровывать малодостоверные суждения?

В нашей модели кликов следует ли просто удалить примеры с низкой достоверностью? Обычно мы не рекомендуем отбрасывать точки данных таким образом.

Фильтрация обучающих данных, таких как точки данных ниже некоторого минимального порога изучений, уменьшает объем обучающих данных, которые у вас есть. Даже при разумном пороге документы для запроса обычно проверяются по степенному закону распределения. Пользователи очень часто проверяют только некоторые документы, в то время как подавляющее большинство проверяют очень редко. Таким образом, порог может удалить слишком много примеров LTR и привести к тому, что модель LTR пропустит важные закономерности. Даже при пороге у вас возникнет проблема, как взвесить примеры со средней достоверностью против примеров с высокой достоверностью, например с запросом *transformers dark of the moon*, приведенным ранее.

Вместо использования жесткого отсечения мы рекомендуем сохранять примеры с низкой достоверностью и просто взвешивать все примеры на основе уровня уверенности. Мы сделаем это с помощью бета-распределения на вычисленных оценках релевантности. Затем мы применим это решение для исправления наших суждений о модели кликов SDBN.

Использование бета-распределения для корректировки достоверности

Бета-распределения¹ помогают нам делать выводы из наших кликов и исследований на основе вероятностей, а не просто предвзятых событий. Однако, прежде чем мы перейдем непосредственно к использованию бета-распределения для суждений, давайте сначала рассмотрим полезность бета-распределения, используя нашу предыдущую, интуитивную аналогию со средним показателем отбивания в бейсболе.

В бейсболе средний показатель отбивания 0.295 для игрока означает, что, когда этот игрок выходит на позицию, есть вероятность примерно 29.5 % вероятности, что он отобьет мяч. Но если бы мы хотели узнать, «каков средний показатель отбивания для этого игрока, отбивающего в Фенуэй-парке в сентябре в дождливые дни», у нас, вероятно, было бы очень мало информации для исхода. Игрок мог отбивать в таких условиях всего несколько раз. Возможно, он отбил 2 мяча из 3 попыток в таких условиях. Мы бы пришли к выводу, что их средний показатель отбивания в этих случаях составляет $2/3$, или 0.67. К настоящему моменту мы знаем, что этот вывод был бы ошибочным: действительно ли мы думаем, основываясь только на 3 случаях отбивания, что можем сделать вывод, что у игрока невероятно высокий шанс 66.7 % сделать хороший удар? Лучшим подходом было бы использовать общий средний показатель отбивания 0.295 в качестве начального убеждения, медленно отходя от этого предположения по мере того, как мы постепенно получаем больше данных об отбиваниях в «Фенуэй Парк в сентябре в дождливые дни».

Бета-распределение – это инструмент, используемый для управления убеждениями. Оно превращает вероятность, такую как средний показатель отбивания или оценка суждения, в два значения, a и b , которые представляют вероятность как распределение. Значения a и b можно интерпретировать следующим образом:

- a (успехи) – количество наблюдаемых ударов с удачными отбиваниями или количество изучений с кликами;
- b (неудачи) – количество наблюдаемых ударов без отбиваний или количество изучений без кликов.

С бета-распределением выполняется свойство $mean = a / (a + b)$, где $mean$ – это начальное значение как средний показатель отбивания. При наличии среднего значения мы можем найти много значений a и b , которые удовлетворяют $mean = a / (a + b)$. В конце концов, $0.295 = 295 / (295 + 705)$, как и $0.295 = 1475 / (1475 + 3525)$ и т. д. Тем не менее

¹ Бета-распределение в теории вероятностей и статистике – это двухпараметрическое семейство абсолютно непрерывных распределений. Используется для описания случайных величин, значения которых ограничены конечным интервалом. Это распределение вероятностей *по вероятностям*. Мы можем использовать его для моделирования вероятностей: рейтинг кликов вашей рекламы, коэффициент конверсии клиентов, фактически купивших что-то на вашем сайте, насколько вероятно, что посетители поставят лайки в вашем блоге, и т.д. – *Прим. ред.*

каждое из них представляет собой разное бета-распределение. Помните об этом свойстве по мере продвижения вперед.

Давайте соберем эти части вместе, чтобы увидеть, как бета-распределение не дает нам делать поспешные выводы на основе ложных данных о кликах (или удачных отбиваниях).

Мы могли бы объявить наше первоначальное убеждение об уровне релевантности любого документа как 0.125. Это все равно, что объявить средний показатель отбивания бейсболиста равным 0.295 в качестве нашего первоначального убеждения о его результативности. Мы можем использовать бета-распределение для обновления первоначального убеждения для конкретных случаев, таких как «Фенуэй Парк в сентябре в дождливые дни» или релевантность конкретного документа для поискового запроса.

Первый шаг – выбрать a и b , которые отражают наше первоначальное убеждение. Для нашего случая релевантности мы могли бы выбрать много значений для a и b , которые удовлетворяют $0.125 = a / (a + b)$. Предположим, мы выбираем $a = 2.5$, $b = 17.5$ в качестве нашего убеждения о релевантности для документов без кликов. Графически изобразив это, мы увидим распределение на рис. 11.8.

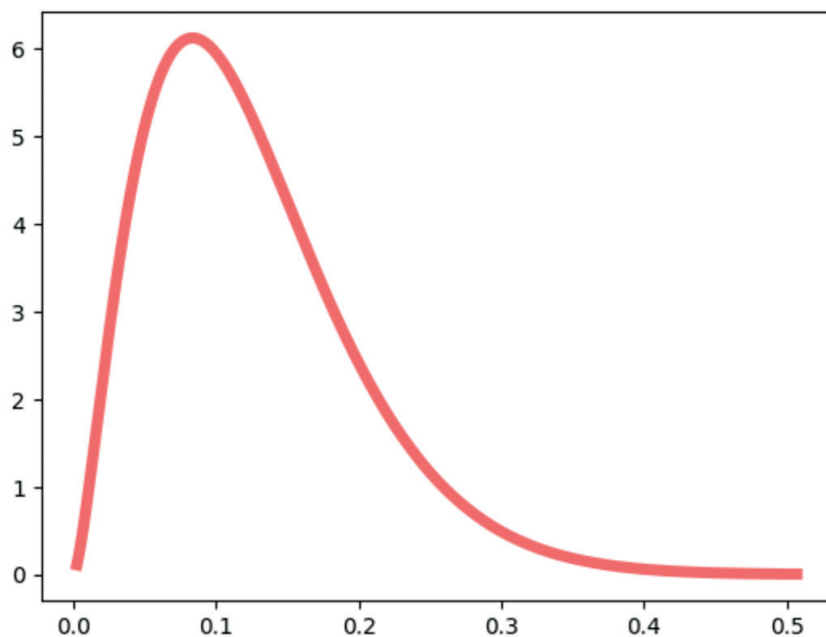


Рис. 11.8. Бета-распределение для степени релевантности 0.125. Среднее значение соответствует нашей степени релевантности по умолчанию. Мы видим распределение наиболее вероятных степеней релевантности

Теперь можно наблюдать, что происходит, когда мы видим первый клик документа, увеличивая a этого документа до 3.5. На рис. 11.9 мы имеем $a=3.5$, $b=17.5$.

Средняя степень релевантности для обновленного распределения теперь составляет $3.5 / (17.5 + 3.5)$, или 0.1667, что фактически немного повышает первоначальное убеждение, учитывая его первый клик. Без бета-распределения этот документ имел бы 1 клик и 1 проверку, что дало бы оценку 1.

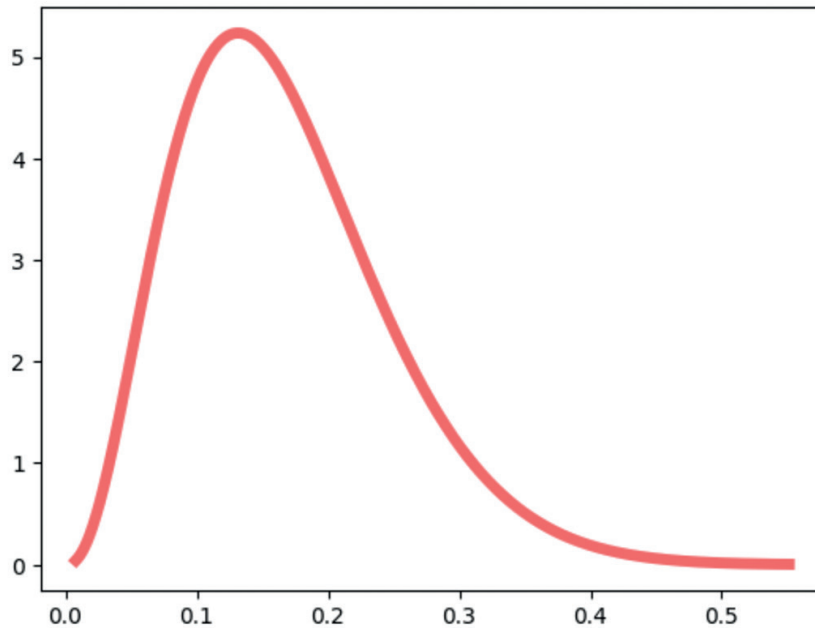


Рис. 11.9. Бета-распределение для степени релевантности после добавления одного клика теперь составляет 0.1667. Добавление клика «смещает» распределение вероятностей немного в одном направлении, обновляя первоначальное убеждение

Мы называем начальное распределение вероятностей (выбранные a и b) *априорным распределением* (prior distribution), или просто *априорным* (prior). Это наше первоначальное убеждение в том, что произойдет. Распределение после обновления a и b для конкретного случая, например документа, является *постериорным распределением* (posterior distribution), или просто *постериорным* (posterior). Это наше обновленное убеждение.

Вспомним, что мы говорили ранее, что можно выбрать много начальных значений a и b . Это имеет значение, так как величина начальных a и b делает наше априорное значение слабее или сильнее. Мы могли бы выбрать любое значение для a и b , где $a / (a + b) = 0.125$. Но обратите внимание, что произойдет, если мы выберем очень маленькое значение $a = 0.25$, $b = 1.75$. Затем мы обновляем a , увеличивая его на 1. Новое ожидаемое значение постериорного распределения равно $1.25 / (1.25 + 1.75)$, или ~ 0.416 . Это серьезный эффект всего за один клик. И наоборот, использование очень высоких значений a и b сделает априорное значение настолько сильным, что оно едва сдвинется с места. При использовании бета-распределения вам нужно будет настроить величину априорного значения, чтобы обновления имели желаемый эффект.

Теперь, когда вы увидели этот удобный инструмент для захвата оценок SDBN на практике, давайте посмотрим, как бета-распределение может помочь решить наши проблемы с достоверностью SDBN.

Использование бета-приорного распределения в моделях кликов SDBN

Давайте закончим главу, обновив модель кликов SDBN с помощью бета-распределения. Если вы используете другие модели кликов, напри-

мер упомянутые ранее в этой главе, вам нужно будет подумать о том, как можно решить проблему уверенности в этих случаях. Бета-распределение может быть полезным инструментом и здесь.

Если вы помните, вывод SDBN представлял собой количество `clicks` и `examined` для каждого документа. В листинге 11.12 мы берем это из листинга 11.11, который вычислял SDBN для запроса `blue ray`. Мы выберем предыдущую оценку 0.3 для использования с нашей моделью SDBN. Это наша оценка по умолчанию, когда у нас нет информации о документе, – возможно, полученная из типичной оценки, которую мы видим в наших суждениях. Затем мы вычислим предыдущее бета-распределение (`prior_a` и `prior_b`), используя эту предыдущую оценку.

Листинг 11.12. Вычисление априорного бета-распределения

```
def calculate_prior(sessions, prior_grade, prior_weight):
    sessions = calculate_grade(sessions)
    sessions["prior_a"] = prior_grade * prior_weight
    sessions["prior_b"] = (1 - prior_grade) * prior_weight
    return sessions
```

Результирующие `a` и `b` удовлетворяют $\text{prior_grade} = \text{prior_a} / (\text{prior_a} + \text{prior_b})$.

Априорная оценка релевантности по умолчанию.

Сколько достоверности следует придавать априорной оценке ($\text{prior_weight} = a + b$).

```
prior_grade = 0.3
prior_weight = 100
query = "blue ray"

sessions = get_sample_sessions(query)
prior_data = calculate_prior(sessions, prior_grade, prior_weight)
print(prior_data)
```

Вывод (усеченный):

doc_id	clicked	examined	grade	prior_a	prior_b
600603132872	1.0	1.0	1.000000	30.0	70.0
827396513927	14.0	34.0	0.411765	30.0	70.0
25192073007	8.0	20.0	0.400000	30.0	70.0
885170033412	6.0	19.0	0.315789	30.0	70.0
...					

В листинге 11.12 с весом 100 вы можете подтвердить, что $\text{prior_grade} = \text{prior_a} / (\text{prior_a} + \text{prior_b})$, или $0.3 = 30 / (30 + 70)$. Это зафиксировало начальное распределение вероятностей для нашей априорной оценки.

В листинге 11.13 нам нужно вычислить постериорное распределение и соответствующую оценку релевантности. Мы делаем это, увеличивая `prior_a` для кликов (наши «успехи») и `prior_b` для изучений без кликов (наши «неудачи»). Наконец, мы вычисляем обновленную оценку как `beta_grade`.

Листинг 11.13. Вычисление постериорного бета-распределения

Обновляет нашу веру в отсутствие релевантности документа, увеличивая с `prior_b` на число изучений без кликов.

Обновляет нашу веру в релевантность документа, увеличивая с `prior_a` на число кликов.

```
def calculate_sdbn(sessions, prior_grade=0.3, prior_weight=100):
    sessions = calculate_prior(sessions, prior_grade, prior_weight)
    sessions["posterior_a"] = (sessions["prior_a"] +
                               sessions["clicked"])
    sessions["posterior_b"] = (sessions["prior_b"] +
                               sessions["examined"] - sessions["clicked"])
    sessions["beta_grade"] = (sessions["posterior_a"] /
                              (sessions["posterior_a"] + sessions["posterior_b"]))
    return sessions.sort_values("beta_grade", ascending=False)

query = "blue ray"
sessions = get_sample_sessions(query)
bluray_sdbn_data = calculate_sdbn(sessions)
print(bluray_sdbn_data)
```

Вычисляет новую оценку из `posterior_a` и `posterior_b`.

Вывод (усеченный):

doc_id	cl	ex	grade	pr_a	pr_b	posterior_a	posterior_b	beta_grade
827396513927	14	34	0.411	30.0	70.0	44.0	90.0	0.328358
25192073007	8	20	0.400	30.0	70.0	38.0	82.0	0.316667
600603132872	1	1	1.000	30.0	70.0	31.0	70.0	0.306931
...								
786936805017	1	14	0.071	30.0	70.0	31.0	83.0	0.271930
36725608511	0	11	0.000	30.0	70.0	30.0	81.0	0.270270
23942972389	0	15	0.000	30.0	70.0	30.0	85.0	0.260870

В выводе листинга 11.13 заголовки столбцов `clicked`, `examined`, `prior_a` и `prior_b` были сокращены, чтобы дать место `cl`, `ex`, `pr_a` и `pr_b`. Обратите внимание на наши новые идеальные результаты для запроса `blue ray` путем сортировки по `beta grade`. Значения бета-оценки остаются ближе к предыдущей оценке 0.3. Примечательно, что наши случаи `Blu-ray` скатились на третий по релевантности слот, при этом ни один клик не поднял оценку намного выше 0.3.

Когда мы повторяем этот расчет суждений для `dryer` и `transformers dark of the moon` на рис. 11.10 и 11.11, мы замечаем, что порядок тот же, но сами оценки остаются ближе к априорной 0.3 в зависимости от нашей уверенности в данных.

Рисунок 11.11 показывает заметно меньшую достоверность саундтрека по сравнению с суждениями SDBN без моделирования уверенности (рис. 11.7). Оценка упала с 0.4806 до 0.4017. Примечательно, что оценка DVD после CD не сильно изменилась, изменившись только с 0.3951 до 0.3673 из-за нашей большей уверенности в этом наблюдении. По мере поступления большего количества наблюдений, вероятно, CD даже снизится в рейтинге, если эта закономерность сохранится.

Скорректированные по достоверности суждения SDBN для q=dryer

	beta_grade	upc	image	name
0	0.3853	856751002097		Practecol - Dryer Balls (2-Pack)
1	0.3748	48231011396		LG - 3.5 Cu. Ft. 7-Cycle High-Efficiency Washer - White
2	0.3158	84691226727		GE - 6.0 Cu. Ft. 3-Cycle Electric Dryer - White
3	0.2946	74108007469		Conair - 1875-Watt Folding Handle Hair Dryer - Blue
4	0.2775	12505525766		Smart Choice - 6' 30 Amp 3-Prong Dryer Cord

Рис. 11.10. Результаты SDBN с бета-коррекцией для dryer. Обратите внимание, что теперь оценки более плотно сосредоточены вокруг априорной оценки 0.3, а некоторые выше или ниже этой априорной оценки

Скорректированные по достоверности суждения SDBN для q=transformers: dark of the moon

	beta_grade	upc	image	name
0	0.5957	97360810042		Transformers: Dark of the Moon - Blu-ray Disc
1	0.4017	400192926087		Transformers: Dark of the Moon - Original Soundtrack - CD
2	0.3673	97363560449		Transformers: Dark of the Moon - Widescreen Dubbed Subtitle - DVD
3	0.3130	97363532149		Transformers: Revenge of the Fallen - Widescreen Dubbed Subtitle - DVD
4	0.2795	93624956037		Transformers: Dark of the Moon - Original Soundtrack - CD

Рис. 11.11. Результаты SDBN с бета-коррекцией для transformers: dark of the moon. На рис. 11.7 мы отметили, что саундтрек показался странно высоким (0.48) по своей степени релевантности, несмотря на меньшее количество кликов, чем фильм на Blu-ray. Теперь мы видим, что релевантность саундтрека ближе к априорному значению 0.4

Большинство ваших запросов не будут похожи на *dryer* или *transformers: dark of the moon*. Они будут больше похожи на *blue ray*. Чтобы осмысленно работать с этими запросами для LTR, вам нужно будет уметь справляться с такими проблемами «малых данных», как низкая достоверность.

У нас начинает появляться более разумный обучающий набор для автоматизации LTR, но еще многое предстоит сделать. В следующей главе мы перейдем к рассмотрению полного цикла обратной связи поиска. Это включает в себя работу над предвзятостью представления. Вспомним, что это предвзятость, при которой пользователи никогда не изучают то, что поиск никогда им не возвращает. Как мы можем добавить наблюдение к автоматизированному циклу обратной связи LTR, чтобы преодолеть предвзятость представления и гарантировать, что наша модель – и, как следствие, суждения – работает так, как ожидалось?

Прежде чем мы рассмотрим эти темы в следующей главе, давайте снова рассмотрим обучение модели LTR от начала до конца, чтобы вы могли поэкспериментировать с тем, что вы узнали на данный момент.

11.4. Изучение ваших обучающих данных в системе LTR

Отличная работа! Вы прошли главы 10 и 11. Теперь у вас есть все необходимое для разработки разумных обучающих данных LTR и обучения модели LTR. Вероятно, вы хотите обучить модель на основе своей работы. Вместо того чтобы повторять обширный код из главы 10 здесь, мы создали блокнот «Сквозное автоматизированное обучение ранжированию» в папке `ch11` кодовой базы книги (`4.end-to-end-auto-ltr.ipynb`). Он позволит вам экспериментировать с LTR на данных RetroTech (рис. 11.12).

В этом блокноте вы можете настроить внутренний механизм LTR – проектирование признаков и создание модели, которая пытается удовлетворительно работать с имеющимися обучающими данными. Вы также можете изучить последствия изменения автоматизированных входных данных для этого механизма: самих обучающих данных. В целом в этом блокноте есть все шаги, о которых вы узнали до сих пор.

- 1 Обработка необработанных данных сеансов кликов в суждения с использованием модели кликов SDBN и априорной бета-версии.
- 2 Преобразование фрейма данных в суждения, которые мы использовали в главе 10.
- 3 Загрузка коллекции признаков LTR для использования с возможностями хранилища признаков поисковой системы.
- 4 Регистрация этих признаков, полученных из поисковой системы, и последующее выполнение попарного преобразования данных в подходящий обучающий набор.
- 5 Обучение модели и загрузка ее в хранилище моделей поисковой системы.
- 6 Поиск и ранжирование с помощью данной модели.

Оценки SDBN с использованием бета-распределения

У нас есть около дюжины запросов, для которых мы смоделировали поток кликов. Здесь мы вычисляем суждения SDBN, используя бета-распределение, по каждому из этих запросов. Код в `generate_training_data` просто повторяет то, что мы делали в этом разделе книги, только для каждого запроса, по которому у нас есть данные.

Затем мы преобразуем их в объект `Judgments`, который мы используем в главе 10.

Во что стоит играть

Изучите свеличину априорного веса (`PRIOR_WEIGHT`), а также конкретную оценку релевантности по умолчанию, `PRIOR_GRADE`. Более сильный `PRIOR_WEIGHT` не сильно уступит `PRIOR_WEIGHT`.

Если вы чувствуете себя более подготовленным, вы можете изучить различные методы вычисления суждений о релевантности на основе этих сеансов, заменив `sessions_to_sdbn` собственной формулой для перевода кликов в суждения.

```
PRIOR_GRADE=0.2  
PRIOR_WEIGHT=10
```

Рис. 11.12. Блокнот, исследующий полную систему LTR. Вы можете взять модель на тест-драйв

Мы предлагаем вам настроить параметры модели кликов и подумать о новых признаках и различных способах достижения окончательной модели LTR, обнаруживая, какие из них, по-видимому, дают наилучшие результаты. Пока вы делаете эту настройку, обязательно подвергайте сомнению свои собственные субъективные предположения по сравнению с тем, что показывают вам данные.

С готовой настройкой мы оставим вас с рис. 11.13, показывающим текущие результаты поиска по запросу `transformers dvd`. Попробуйте здесь другие запросы. Как вы можете помочь модели лучше различать релевантные и нерелевантные документы? Являются ли проблемы, с которыми вы сталкиваетесь, следствием используемых обучающих данных? Или они связаны с признаками, используемыми для построения модели?

В следующей главе мы завершим автоматизированную систему LTR, выполнив наблюдение за моделью. Самое главное – мы рассмотрим, как преодолеть *предвзятость представления*. Даже с корректировками в этой главе пользователи по-прежнему будут иметь возможность действовать только на основе того, что им показывает поиск. Таким образом, у нас все еще есть цикл обратной связи, сильно смещенный текущим рейтингом релевантности. Как мы можем отслеживать эту проблему и преодолевать ее? В следующей главе мы рассмотрим эти проблемы, поскольку наша модель LTR продолжает итеративно включать входящие взаимодействия пользователей и активно выводить дополнительные многообещающие результаты.

**Name:** Transformers - DVD**Manufacturer:****Name:** Nintendo - Transformers 3 Stylus 2-Pack**Manufacturer:** Nintendo**Name:** Nintendo - Transformers 3 Cybertanium Case**Manufacturer:** Nintendo

Рис. 11.13. Как наша обученная модель ранжирует transformers dvd. Как вы думаете, вы могли бы улучшить это?

Резюме

- Мы можем автоматизировать обучение ранжированию (LTR), если сможем надежно преобразовать данные о кликах пользователя в суждения о релевантности с помощью модели кликов. Однако модель кликов должна быть тщательно спроектирована и отлажена, чтобы уменьшить предвзятость в данных и обеспечить надежность автоматизированной системы LTR при развертывании для реальных пользователей.
- Изученные (неявные) списки суждений о релевантности можно подключить к существующим процессам обучения LTR, чтобы заменить или дополнить вручную созданные суждения.
- Необработанные клики обычно проблематичны в автоматизированных моделях LTR из-за распространенных предвзятостей в том, как алгоритмы ранжируют и представляют результаты поиска пользователям.
- Среди видимых результатов поиска предвзятость позиции говорит о том, что пользователи предпочитают результаты, ранжированные близко к верхним. Мы можем преодолеть предвзятость позиции, используя модель кликов, которая отслеживает вероятность того, что пользователь изучил документ или данную позицию в полученных им результатах поиска.
- Большинство поисковых приложений имеют много ложных данных кликов. Когда данные обучения смещены в сторону этих ложных результатов, мы получаем предвзятость уверенности. Мы можем преодолеть предвзятость уверенности, используя бета-распределение для создания априорной информации, которую мы постепенно обновляем новыми наблюдениями по мере их поступления.

12

Преодоление предвзятости ранжирования с помощью активного обучения

В этой главе рассматривается:

- использование живых взаимодействий с пользователем для сбора отзывов о развернутой модели LTR;
- А/В-тестирование решений по релевантности поиска с живыми пользователями;
- использование активного обучения для нахождения результатов, потенциально более релевантных, чем верхние результаты;
- баланс между использованием пользовательских взаимодействий и поиском того, что еще может быть релевантным.

До сих пор наша работа по обучению ранжированию (LTR) проходила в лаборатории. В предыдущих главах мы строили модели, используя автоматически сконструированные обучающие данные из кликов

пользователей. В этой главе мы перенесем нашу модель в реальный мир для тест-драйва с (имитированными) живыми пользователями!

Напомним, что мы сравнивали автоматизированную систему LTR с беспилотным автомобилем. Внутри у автомобиля есть двигатель: сквозная модель, переобучающаяся на исторических суждениях, как обсуждалось в главе 10. В главе 11 мы сравнили данные обучения нашей модели с указаниями беспилотному автомобилю: что нам *следует* оптимизировать, чтобы автоматически изучать суждения на основе предыдущих взаимодействий с результатами поиска? Мы создали данные обучения и преодолели ключевые предвзятости, присущие данным кликов.

В этой главе мы сосредоточимся на перемещении наших моделей ранжирования из лаборатории в производство. Мы развернем и будем отслеживать наши модели по мере получения ими пользовательского трафика. Мы увидим, где модель работает хорошо, и поймем, была ли работа в предыдущих двух главах успешной или неудачной. Это означает изучение нового вида тестирования для проверки нашей модели: *A/B-тестирование*. В A/B-тестировании мы случайным образом назначаем живых пользователей разным моделям и изучаем бизнес-результаты (продажи и т. д.), чтобы увидеть, какие модели работают лучше всего. Вы, возможно, знакомы с A/B-тестированием в других контекстах, но здесь мы сосредоточимся на последствиях для автоматизированной системы LTR. Живые пользователи не только помогают нам проверить нашу систему, но и помогают избежать опасных циклов отрицательной обратной связи, в которых могут оказаться наши модели, как показано на рис. 12.1.



Рис. 12.1. Цикл отрицательной обратной связи предвзятости представления. Пользователи никогда не кликают на то, что поисковая система никогда не возвращает, поэтому модели релевантности никогда не могут выйти за рамки знаний текущей модели

На рис. 12.1 наша модель может узнать только то, что релевантно *в результатах, показанных пользователю*. Другими словами, у нас есть досадная проблема курицы и яйца: модель обучается на основе того, что пользователи считают релевантным, но то, что пользователи считают релевантным, основано на том, что показывает им модель. Хороший LTR пытается оптимизировать результаты с наиболее положительными сигналами взаимодействия, но пользователи будут кликать только на то, что находится прямо перед ними. Как LTR может улучшиться, если данные обучения кажутся безнадежно предвзятыми в сторону текущего рейтинга поисковой системы? Эта предвзятость неявно полученных данных обучения, отражающих ранее отображаемые результаты, называется *предвзятостью представления*.

После того как мы рассмотрим А/В-тестирование, мы будем бороться с предвзятостью представления до конца главы с помощью *активного обучения*. Система активного обучения – это система, которая может интерактивно собирать новые маркированные данные, полученные от анализа действий пользователей, чтобы отвечать на новые вопросы. В нашем случае наш алгоритм активного обучения определит слепые зоны, ведущие к предвзятости ранжирования, предложит пользователям взаимодействовать с новыми результатами, исследуя эти слепые зоны, и будет использовать интерактивные взаимодействия с пользователем в качестве новых данных обучения для исправления слепых зон. Подобно беспилотному автомобилю, который изучил только один неоптимальный путь, нам придется стратегически исследовать альтернативные перспективные пути – в нашем случае дополнительные типы результатов поиска, – чтобы изучить новые шаблоны для того, что актуально для пользователей. На рис. 12.2 мы видим автоматизированный цикл LTR, дополненный этим исследованием слепых зон. Прежде чем мы приступим к этой крайне важной теме, нужно сначала обернуть все, что мы узнали в главах 10 и 11, в несколько строк кода. Затем мы сможем быстро итерировать, исследуя А/В-тестирование и преодолевая предвзятость представления.

12.1. Наш автоматизированный движок LTR в нескольких строках кода

Прежде чем приступить к А/В-тестированию, давайте соберем все наши знания из глав 10 и 11 в небольшую горстку многоцветных вспомогательных функций Python. Сначала мы определим функцию, которая позволит нам перестроить обучающие данные из необработанных кликов сеанса с использованием упрощенной динамической байесовской сети (SDBN) модели кликов (вся глава 11). Затем мы создадим столь же простой фрагмент кода для обучения модели с этими обучающими данными (вся глава 10). Мы очень быстро суммируем эти функции, прежде чем погрузимся в А/В-тестирование и преодоление предвзятости представления в следующей части главы.

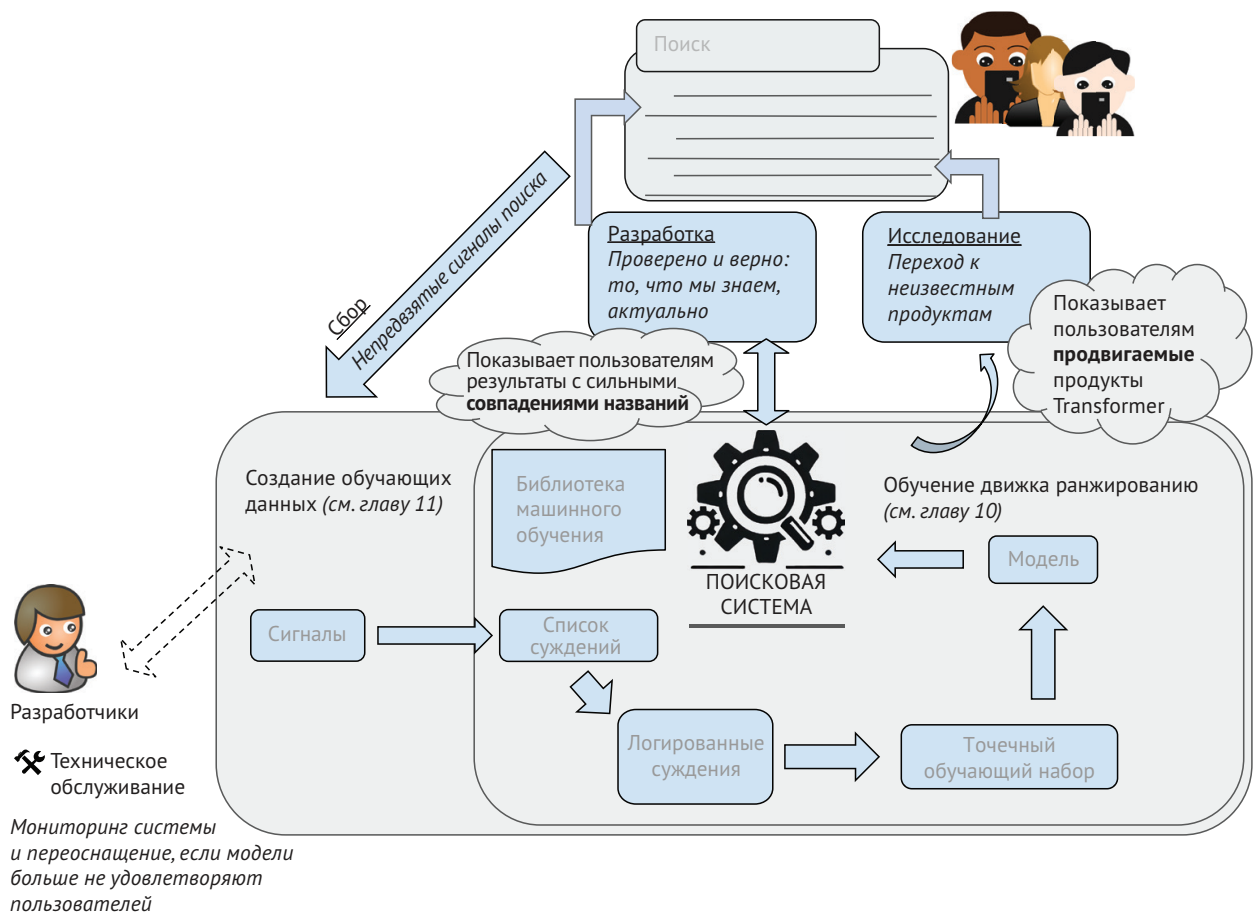


Рис. 12.2. Автоматизированный LTR встречается реальных пользователей. Чтобы быть полезной, наша автоматизированная система LTR должна преодолеть предвзятость представления, изучая еще не полученные результаты с пользователями, чтобы расширить охват обучающих данных

12.1.1. Превращение кликов в обучающие данные (глава 11 в одной строке кода)

В главе 11 мы превратили клики в обучающие данные и изучили модель кликов SDBN, которая может преодолеть предвзятость пользователей в их реакции на результаты поиска. Мы повторно используем большую часть кода из главы 11, поскольку исследуем дополнительные предвзятости и автоматизируем сквозной процесс LTR в этой главе.

Напоминаем, что наша модель кликов превращает необработанные клики в обучающие метки или *оценки*, отображающие, насколько релевантным является документ для ключевого слова. Необработанные входные данные, необходимые нам для построения обучающих данных, включают строку запроса, ранг отображаемого результата, документ в этой позиции и был ли он кликнут. Мы видим эти данные в следующем фрейме данных:

id	sess_id	query	rank	doc_id	clicked
0	50002	blue ray	0.0	600603141003	True
1	50002	blue ray	1.0	827396513927	False

2	50002	blue ray	2.0	24543672067	False
3	50002	blue ray	3.0	719192580374	False
4	50002	blue ray	4.0	885170033412	True

Учитывая эти входные данные, мы можем упаковать всю главу 11 в повторно используемую функцию, которая вычисляет наши обучающие данные. Напомним, что мы используем термин *список суждений* или *суждения* для обозначения наших обучающих данных. Мы можем увидеть вычисление наших суждений в следующем листинге.

Листинг 12.1. Генерация обучающих данных из сеансов (глава 11)

```
training_data = generate_training_data(sessions, prior_weight=10,
                                      prior_grade=0.2)
display(training_data)
```

Вывод (усеченный):

query	doc_id	clicked	examined	grade	beta_
blue ray	27242815414	42	42	1.000000	0.846154
	827396513927	1304	3359	0.388211	0.387652
	883929140855	140	506	0.276680	0.275194
	885170033412	568	2147	0.264555	0.264256
	24543672067	665	2763	0.240680	0.240534
...					
transformers	dvd 47875819733	24	1679	0.014294	0.015394
	708056579739	23	1659	0.013864	0.014979
	879862003524	23	1685	0.013650	0.014749
	93624974918	19	1653	0.011494	0.012628
	47875839090	16	1669	0.009587	0.010721

Функция `generate_training_data` принимает все сеансы поиска пользователя вместе с `prior_weight`, указывающим, насколько сильно должно быть взвешено априорное значение (по умолчанию 1.0), и `prior_grade`, указывающим вероятность релевантности результата по умолчанию, когда у нас нет доказательств (по умолчанию 0.2). Смотрите раздел 11.3.2 для освежения в памяти того, как эти значения влияют на расчет SDBN.

Давайте кратко повторим то, что мы узнали в главе 11, посмотрев на листинг 12.1. Как вы можете видеть в выводе, мы вычисляем датафрейм, где каждая пара запрос–документ имеет соответствующие счетчики `clicked` и `examined`. Клики – это то, как они звучат: сумма не обработанных кликов, которые данный продукт получил для данного запроса. Напомним, что `examined` соответствует количеству раз, когда модель кликов считает, что пользователь обратил внимание на результат («изучил» его).

Статистические оценки `grade` и `beta_grade` – это обучающие метки. Они соответствуют вероятности того, что документ соответствует запросу. Вспомните, что оценка просто делит `clicked` на `examined`: наивная

первая реализация модели кликов SDBN. Однако в главе 11 мы узнали, что лучше учитывать, сколько у нас информации (см. раздел 11.3). Мы не хотим, чтобы один клик с одним изучением ($1 / 1 = 1.0$) учитывался так же сильно, как сто кликов с сотней изучений ($100 / 100 = 1.0$). По этой причине `beta_grade` придает больший вес результатам с большим количеством информации (предпочитая пример со ста кликами). Поэтому мы будем использовать `beta_grade` вместо `grade` при повторном обучении моделей LTR.

Эти данные служили данными для обучения моделей LTR, которые мы обучали в главе 10. Далее давайте посмотрим, как мы можем легко взять эти данные для обучения, обучить модель и развернуть ее.

12.1.2. Обучение и оценка модели в нескольких вызовах функций

В дополнение к повторной генерации данных обучения нам также необходимо переобучить нашу модель перед ее запуском для реальных пользователей. В этом разделе мы рассмотрим удобные функции для нашего основного движка обучения модели LTR. Это позволит нам быстро экспериментировать с моделями в оставшейся части этой главы.

Мы упакуем обучение модели и офлайн-оценку в несколько простых строк.

Листинг 12.2. Обучение и оценка модели по нескольким признакам

```
def train_and_evaluate_model(sessions, model_name, features, log=False):
    training_data = generate_training_data(sessions)
    train, test = split_training_data(training_data, 0.8)
    train_and_upload_model(train, model_name, features=features, log=log)
    evaluation = evaluate_model(test, model_name, training_data, log=log)
    return evaluation
```

```
feature_set = [
    ltr.generate_query_feature(feature_name="long_description_bm25",
                              field_name="long_description"),
    ltr.generate_query_feature(feature_name="short_description_constant",
                              field_name="short_description",
                              constant_score=True)]
```

```
evaluation = train_and_evaluate_model(sessions, "ltr_model_variant_1",
                                     feature_set)
```

```
display(evaluation)
```

Оценка для `ltr_model_variant_1`:

```
{"dryer": 0.03753076750950996,
 "blue ray": 0.0,
 "headphones": 0.0846717500031762,
 "dark of moon": 0.0,
 "transformers dvd": 0.0}
```


С помощью листинга 12.2 давайте кратко рассмотрим то, что мы узнали в главе 10. Мы определяем `feature_set` с двумя признаками для LTR: один для поиска по полю `long_description`, а другой для поиска по полю `short_description`. Мы должны выбирать тщательно, надеясь найти признаки, которые осмысленно предсказывают релевантность и которые можно извлечь из обучающих данных в листинге 12.1. Затем разделяем `training_data` на наборы `train` и `test` и используем набор `train` для обучения и загрузки модели.

Но как мы узнаем, успешно ли наша модель обучилась на обучающих данных? Разделение суждений и исключение набора `test` во время обучения модели оставляет часть обучающих данных для оценки обученной модели. Вы как профессор, дающий студенту (здесь модели) выпускной экзамен. Вы можете дать студентам много примеров задач для изучения к тесту (набор `train`). Но чтобы увидеть, действительно ли студенты усвоили материал, а не просто заучили его, вы бы дали им финальное задание с другими вопросами (набор `test`). Это поможет вам оценить, понимает ли ученик то, чему вы его научили, прежде чем отправлять его в реальный мир.

Конечно, успех в классе не всегда равен успеху в реальном мире. Переход нашей модели в реальный мир с живыми пользователями в A/B-тесте может показать, что она работает не так хорошо, как мы надеялись.

Наконец, какая статистика рядом с каждым тестовым запросом? Как мы оцениваем успех учеников в тестовых запросах? Вспомните из главы 10, что мы просто использовали точность (долю релевантных запросов). Эта статистика суммирует топ-N оценок и делит на N (для нас $N = 10$), что фактически является средней оценкой релевантности. Мы рекомендуем изучить другие статистики для обучения и оценки модели, которые смещены в сторону получения правильных верхних позиций, такие как дисконтированный кумулятивный прирост¹ (DCG), нормированный DGC (NDCG) или ожидаемый обратный ранг² (ERR). Для наших целей мы остановимся на более простой статистике точности.

Если судить только по показателям релевантности для наших тестовых запросов в листинге 12.2, наша модель довольно плохо справляется с офлайн-тестированием. Улучшив офлайн-показатели, мы должны увидеть значительное улучшение с живыми пользователями в A/B-тесте.

¹ Дисконтированный кумулятивный прирост измеряет качество ранжирования. Он суммирует полезность результатов, дисконтированную по их позиции в списке результатов. – *Прим. ред.*

² Ожидаемый обратный ранг, англ. *Expected Reciprocal Rank*, – это метрика качества ранжирования, которая показывает, с какой вероятностью среди первых K элементов в упорядоченном наборе найдется тот, который окажется полезным для пользователя (т. е. пользователь остановится на нем). Она основана на каскадной модели поиска, в которой пользователь просматривает результаты в порядке ранжирования и останавливается на первом документе, который удовлетворяет информационную потребность. ERR полезна в сценариях, где первостепенной задачей является удовлетворение пользователя. – *Прим. ред.*

12.2. А/В-тестирование новой модели

В этом разделе мы смоделируем запуск А/В-теста и сравним модель листинга 12.2 с моделью, которая, по-видимому, работает лучше в лаборатории. Мы поразмышляем над результатами А/В-теста, настроив нас на завершение автоматизированного цикла обратной связи LTR, который мы представили в главе 11. Мы закончим размышлениями о том, что пошло не так хорошо, потратив оставшуюся часть главы на добавление «активного обучения», важнейшего, недостающего элемента в наш автоматизированный цикл обратной связи LTR.

12.2.1. Выбираем лучшую модель для тестирования

Наша исходная модель LTR показала себя не очень хорошо, как мы увидели в выводе листинга 12.2. В этом разделе мы обучим новую модель, и, как только она покажется многообещающей, мы развернем ее в А/В-тесте против модели, которую обучили в листинге 12.2.

Давайте рассмотрим следующую улучшенную модель.

Листинг 12.3. Новая модель, улучшенная путем изменения признаков

```
feature_set = [
    ltr.generate_fuzzy_query_feature(feature_name="name_fuzzy",
                                     field_name="name"),
    ltr.generate_bigram_query_feature(feature_name="name_bigram",
                                     field_name="name"),
    ltr.generate_bigram_query_feature(feature_name="short_description_bigram",
                                     field_name="short_description")]

evaluation = train_and_evaluate_model(sessions, "ltr_model_variant_2",
                                     feature_set)
```

display(evaluation)

Оценка для ltr_model_variant_2:

```
{"dryer": 0.07068309073137659,          #Before: 0.038
 "blue ray": 0.0,                      # 0.0
 "headphones": 0.06540945492120899,    # 0.085
 "dark of moon": 0.2576592004029579,    # 0.0
 "transformers dvd": 0.10077083021678328} # 0.0
```

В предыдущем листинге мы определяем `feature_set`, содержащий три признака: `name_fuzzy`, который выполняет нечеткий поиск по полю `name`, `name_bigram`, который выполняет поиск фразы из двух слов по полю `name`, и `short_description_bigram`, который выполняет поиск фразы из двух слов по полю `short_description`. Как и прежде, эта модель обучается, развертывается и оценивается. Обратите внимание на вывод листинга 12.3 – на том же наборе тестовых запросов наша модель, по-

хоже, работает намного лучше. Это кажется многообещающим! Действительно, мы выбрали набор признаков, который, кажется, лучше отражает аспекты соответствия текста релевантности.

Проницательный читатель может заметить, что мы сохранили тестовые запросы такими же, как в листинге 12.2. Мы намеренно сделали это для ясности. Этого достаточно, чтобы научить вас фундаментальным навыкам поиска с использованием ИИ. Однако в реальной жизни нам бы хотелось по-настоящему случайного разделения тестовых и тренировочных данных, чтобы лучше оценить производительность модели. Мы могли бы даже пойти дальше, выполнив *перекрестную проверку* (cross-validation) – повторную выборку и обучение многих моделей на разных разделениях тестовых и тренировочных наборов данных, чтобы гарантировать, что модели хорошо обобщаются без переобучения новыми обучающими данными. Если вы хотите глубже погрузиться в оценку офлайн-моделей, мы рекомендуем более общую книгу по машинному обучению, например «Машинное обучение. Портфолио реальных проектов» Алексея Григорьева (Manning, 2021).

Возможно, ваша поисковая группа считает, что обученная в листинге 12.3 модель перспективна и достаточно хороша для развертывания в производстве. Надежды команды возросли, поэтому давайте посмотрим, что произойдет, когда мы запустим разработку в эксплуатацию для дальнейшей оценки с участием реальных пользователей.

12.2.2. Определение A/B-теста в контексте автоматизированного LTR

К концу главы 11 мы разработали сквозной процесс переобучения LTR: мы могли принимать входящие пользовательские сигналы для генерации модели кликов, использовать модель кликов для генерации суждений, использовать суждения для обучения модели LTR, а затем развернуть модель LTR в производстве, чтобы собрать больше сигналов для перезапуска процесса. С помощью этого цикла переобучения LTR мы можем легко развернуть многообещающие новые модели ранжирования.

Однако мы еще не развернули наши модели LTR в производстве. Мы разработали только теоретические модели. Как мы узнаем, хорошо ли то, что мы построили в лаборатории, работает в реальном мире? Совсем другое дело – обрабатывать реальные сценарии.

В этом разделе мы рассмотрим результаты A/B-тестирования с (имитированными) живыми пользователями. Поскольку это книга с кодовой базой, которую вы запускаете локально, к сожалению, мы не можем заставить реальных пользователей обращаться к нашему приложению. Таким образом, «живой» трафик пользователей, который мы будем использовать, будет просто смоделирован из нашей кодовой базы. Для наших целей эта симуляция трафика достаточно похожа на живые взаимодействия пользователей, чтобы успешно продемонстрировать процесс активного обучения.

Мы увидим, как А/В-тест служит в качестве окончательного арбитра успеха нашей автоматизированной системы LTR. Он позволит нам исправить проблемы в нашей автономной автоматизированной модели обучения LTR, чтобы цикл обратной связи мог постепенно становиться более надежным.

Вы можете знать о А/В-тестах, но здесь мы покажем, как они влияют на автоматизированную систему LTR. Как показано на рис. 12.3, *А/В-тест* случайным образом назначает пользователей двум вариантам. Каждый *вариант* содержит отдельный набор признаков приложения. Это может включать в себя все, от разных цветов кнопок до новых алгоритмов ранжирования релевантности. Поскольку пользователи случайным образом назначаются вариантам, мы можем более надежно сделать вывод о том, какой вариант лучше всего работает для выбранных бизнес-результатов, таких как продажи, время, проведенное в приложении, удержание пользователей или что-либо еще, что бизнес может выбрать для приоритета.

При запуске А/В-теста вы обычно делаете один из вариантов *контрольной группой*, представляющей текущий алгоритм по умолчанию. Наличие контроля позволяет вам измерять улучшение других моделей. Также часто выполняется многовариантное тестирование, при котором одновременно тестируется несколько вариантов или их комбинации. Можно реализовать более продвинутые стратегии тестирования, например тестирование с многоруким бандитом¹, когда тест постоянно смещает живой трафик в сторону наиболее эффективных в данный момент вариантов, или бэкестинг на основе сигналов², когда вы используете исторические данные для моделирования А/В-теста, чтобы предсказать лучший вариант в автономном режиме, прежде чем показывать результаты реальным пользователям.

12.2.3. Перевод лучшей модели в А/В-тест

Далее мы задействуем нашу многообещающую новую модель `ltr_model_variant_2` из листинга 12.3 в А/В-тесте. Затем рассмотрим последствия результатов теста. Надежды велики, и ваша команда считает, что эта модель может превзойти конкурентов: плохо работающую `ltr_model_variant_1` из листинга 12.2.

¹ Тестирование с многоруким бандитом, англ. *multi-arm bandit testing*, – это альтернатива классическим А/В-тестам, которая позволяет динамически распределять трафик: более эффективные вариации со временем получают больше трафика, а менее эффективные – меньше. – Прим. ред.

² Бэкестинг на основе сигналов – это метод оценки эффективности торговой стратегии с помощью исторических рыночных данных, в котором используются сигналы для принятия решений. – Прим. ред.

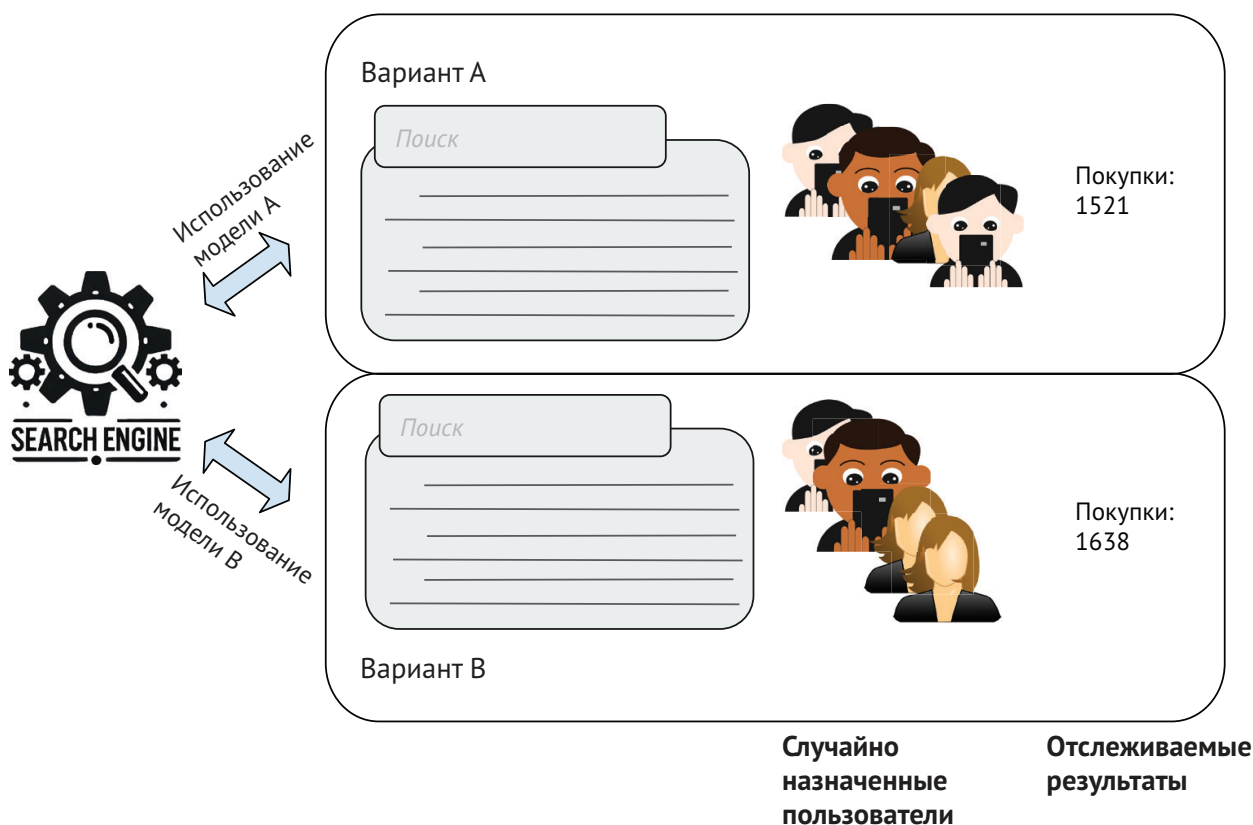


Рис. 12.3. Поисковый А/В-тест. Пользователи поиска случайным образом назначаются двум алгоритмам релевантности (здесь две модели LTR) с отслеживаемыми результатами

В этом разделе мы смоделируем А/В-тест, случайным образом назначая 1000 пользователей каждой модели. В нашем случае эти смоделированные пользователи хотят купить определенные продукты. Если они видят эти продукты, они совершают покупку и покидают наш магазин довольными. Если нет, они могут просмотреть список, но, скорее всего, уйдут, не совершив покупку. Наша поисковая команда, конечно, не знает, что пользователи надеются купить, – эта информация скрыта от нас. Мы видим только поток кликов и покупок, который, как мы увидим, сильно зависит от предвзятости представления.

В листинге 12.4 у нас есть популяция пользователей, ищущих новейшие фильмы франшизы «Трансформеры», выполняя поиск по запросу `transformers dvd`. Мы сосредоточимся на этом единственном запросе во время нашего обсуждения. Конечно, при реальном А/В-тестировании мы бы просмотрели весь набор запросов, и популяция пользователей не была бы такой статичной. Однако, сосредоточившись на одном запросе, мы можем более конкретно понять последствия нашего А/В-теста для автоматизированного LTR. Для более глубокого обзора хорошего эксперимента с А/В-тестированием мы рекомендуем книгу «Эксперименты для инженеров: от А/В-тестирования до байесовской оптимизации» Дэвида Свита (Manning, 2023).

Для каждого запуска функции `a_b_test` в листинге 12.4 модель назначается случайным образом. Затем функция `simulation_live_user_session` имитирует пользователя, выполняющего поиск с запросом и выбранной мо-

делью, сканирующего результаты, возможно, делающего клик и совершающего покупку. Неизвестно нам, что у нашей пользовательской популяции есть скрытые предпочтения за их запросами, которые имитируются в `simulation_live_user_session`. Мы запускаем `a_b_test` 1000 раз, собирая покупки, сделанные пользователями, которые используют каждую модель.

Листинг 12.4. Имитация А/В-теста для запроса `transformers dvd`

```
def a_b_test(query, model_a, model_b):
    draw = random.random()
    model_name = model_a if draw < 0.5 else model_b
    purchase_made = simulate_live_user_session(query, model_name)
    return (model_name, purchase_made)

def simulate_user_a_b_test(query, model_a, model_b, number_of_users=1000):
    purchases = {model_a: 0, model_b: 0}
    for _ in range(number_of_users):
        model_name, purchase_made = a_b_test(query, model_a, model_b)
        if purchase_made:
            purchases[model_name] += 1
    return purchases

results = simulate_user_a_b_test("transformers dvd",
                                "ltr_model_variant_1",
                                "ltr_model_variant_2")

display(results)
```

Случайно назначает каждому пользователю модель а или b.

Имитирует поведение пользователя при поиске и покупке.

Имитирует number_of_users, проходящих тестирование.

Подсчитывает общее количество покупок, сделанных каждой моделью.

Вывод:

```
{"ltr_model_variant_1": 21,
 "ltr_model_variant_2": 15}
```

Как мы видим в выводе листинга 12.4, `ltr_model_variant_2` (наш золотой ученик) на самом деле показывает худшие результаты в этом А/В-тесте! Как это может быть? Что могло пойти не так, чтобы у него были такие хорошие показатели метрики офлайн-теста, но плохие результаты в реальном мире?

В оставшейся части этой главы мы погрузимся в то, что происходит, и попытаемся решить проблему. Таким образом, вы узнаете, как реальные пользователи могут повысить точность вашей автоматизированной системы LTR, что позволит вам с достоверностью переобучиться!

12.2.4. Когда «хорошие» модели работают плохо: чему мы можем научиться из неудачного А/В-теста

Как мы увидели в листинге 12.4, многое может измениться, когда наша модель попадает в реальный мир. В этом разделе мы рассмотрим последствия только что проведенного А/В-теста, чтобы понять, какие дальнейшие шаги будут целесообразны.

Когда модель хорошо работает в лаборатории, но не проходит A/B-тест, это означает, что, хотя мы и построили «правильную» модель LTR, мы построили ее по неправильной спецификации. Нам нужно исправить проблемы с самими данными обучения: суждениями, полученными из нашей модели кликов.

Но как проблемы могут проникнуть в наши суждения, основанные на модели кликов? Мы увидели две проблемы в главе 11: *предвзятость позиции* и *предвзятость уверенности*. В зависимости от ваших целей, UX¹ и домена могут появиться дополнительные смещения. В электронной коммерции пользователи могут соблазниться щелкнуть по продукту, который продается по распродаже, что приведет к искажению данных в сторону этих продуктов. В исследовательской обстановке одна статья может предоставить более богатое резюме в результатах поиска, чем другая. Некоторые предвзятости стирают грань между «предвзятостью» и фактической релевантностью для этого домена. Например, продукт без изображения может получить меньше кликов. Он может быть технически идентичен другому «релевантному» продукту с изображением, но для пользователей продукт без изображения кажется менее надежным и, таким образом, не будет кликнут. Является ли это предвзятостью или просто фактическим показателем релевантности для этого домена, где надежность продукта является фактором?

Чтобы принимать более обоснованные решения, следует ли игнорировать или обесценивать клики, и следует ли вместо этого использовать другие поведенческие сигналы? Возможно, нужно включить последующие действия после клика, такие как клик кнопки «нравится», добавление продукта в корзину или клик кнопки «подробнее»? Возможно, нам следует игнорировать «дешевые» или случайные клики, когда пользователь сразу же нажимает кнопку «назад» после клика?

Рассмотрение действий после клика может быть ценным. Однако мы должны спросить, насколько сильно рейтинг поиска влияет на такие события, как покупка или добавление в корзину, или они связаны с другими факторами. Например, отсутствие покупок может указывать на проблему со страницей отображения продукта или со сложным процессом оформления заказа, а не только с релевантностью результата поиска для конкретного запроса.

Мы можем использовать такой результат, как общее количество покупок, в совокупности по всем запросам для каждой тестовой группы, чтобы оценить результаты A/B-теста. Пока все остальные переменные в приложении остаются неизменными, за исключением алгоритма ранжирования, мы знаем, что любая существенная разница в покупках в разных тестовых группах должна быть вызвана одной вещью, которую мы изменили. Однако в конкретной связи запрос–документ причинно-следственная связь усложняется. Любой отдельный про-

¹ UX в программировании связано с пользовательским опытом (user experience). Это то, каким образом пользователь взаимодействует с интерфейсом и насколько сайт или приложение для него удобны. – Прим. ред.

дукт может иметь очень мало покупок (многие люди смотрят телевизор за 2000 долл., но очень немногие покупают). Данных может просто не хватать для того, чтобы узнать, связана ли покупка исключительно с конкретной релевантностью продукта для запроса. Учет всех вариаций в поисковом UX, доменах и поведении занял бы много книг и все равно был бы недостаточным. Пространство поиска постоянно развивается, и новые способы взаимодействия с результатами поиска входят в моду или устаревают. Для большинства сценариев достаточно будет использования кликов и стандартных моделей кликов. Клики в поисковых UI¹ были тщательно изучены. Тем не менее принятие правильных решений – это и искусство, и наука; вы можете обнаружить, что небольшая модификация модели кликов, которая учитывает дополнительные сигналы, важна для вашего домена и может обеспечить огромные преимущества в том, как ваша модель работает в A/B-тесте. Вы можете потратить столько же времени на совершенствование своей модели кликов, сколько и на разработку своей поисковой системы.

Однако есть одна универсальная пагубная проблема с данными обучения, которая бросает вызов всем моделям кликов: *предвзятость представления*. Предвзятость представления возникает, когда наши модели не могут узнать, что релевантно из кликов пользователей, потому что результаты никогда не появляются для кликов изначально. Далее мы углубимся в эту сложную проблему и узнаем, как преодолеть эту предвзятость, одновременно оптимизируя как то, чему наши модели уже научились, так и то, чему им еще предстоит научиться.

12.3. Преодоление предвзятости представления: знание того, когда исследовать, а когда эксплуатировать

Независимо от того, используем ли мы клики или более сложные сигналы, пользователи никогда не взаимодействуют с тем, чего они не видят. Другими словами, в основе автоматизированного LTR лежит проблема курицы и яйца: если релевантный результат никогда не возвращается исходной, плохо настроенной системой, как любая система машинного обучения на основе кликов может узнать, что результат релевантен?

В этом разделе вы узнаете об одной методике машинного обучения, которая выбирает документы для исследования, *несмотря на то что эти результаты не содержат данных о кликах*. Эта последняя недостающая часть нашей автоматизированной системы LTR помогает нам не только строить модели, оптимизированные для обучающих данных, но и активно участвует в собственном обучении, чтобы расширить широту доступных обучающих данных. Мы называем систему, которая участвует в собственном обучении, *активной обучающей системой*. Рисунок 12.4 демонстрирует

¹ UI (User Interface) в программировании – это пользовательский интерфейс, то, с чем взаимодействует пользователь. – Прим. ред.

предвзятость представления. Предметы в правой части рисунка могли бы быть релевантными для нашего запроса, но без трафика у нас нет данных, чтобы узнать об этом. Было бы неплохо направить трафик на эти результаты, чтобы узнать, считают ли их релевантными пользователи.

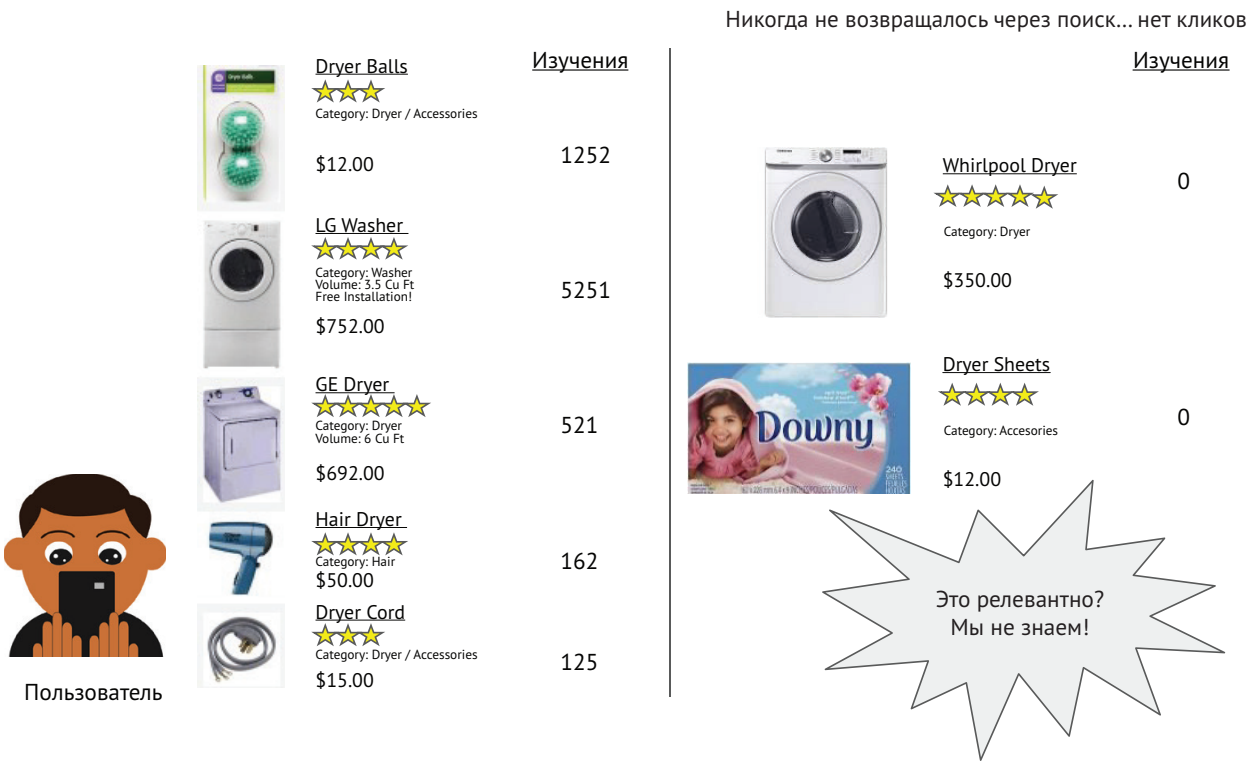


Рис. 12.4. Предвзятость представления (по сравнению с неисследованными результатами) для запроса dryer

Чтобы преодолеть предвзятость представления, мы должны тщательно сбалансировать *использование* текущих, с трудом заработанных знаний нашей модели и исследование за пределами этих знаний. Это компромисс между *исследованием* и *эксплуатацией*. Исследование позволяет нам получать знания, расширяя охват нашей модели кликов для новых и различных типов документов. Однако, если мы всегда будем исследовать, мы никогда не воспользуемся нашими знаниями. Когда мы эксплуатируем, мы оптимизируем то, что, как мы знаем, работает хорошо. Эксплуатация соответствует нашей текущей модели LTR, которая согласуется с нашими обучающими данными. Знание того, как систематически балансировать исследование и эксплуатацию, является ключевым, и это то, что мы обсудим в следующих нескольких разделах, используя инструмент машинного обучения (гауссов процесс¹), созданный для этой цели.

¹ Гауссов процесс в программировании – это непараметрический метод контролируемого обучения, используемый для решения задач регрессии и вероятностной классификации. Он построен на двух основных концепциях: средней функции, которая представляет среднее предсказание, и ковариационной функции, также известной как ядро, которая определяет, как точки в наборе данных соотносятся друг с другом. С помощью ядер гауссовы процессы могут обрабатывать нелинейности, моделировать сложные взаимосвязи и генерировать прогнозы путем экстраполяции и интерполяции данных из наблюдаемых точек. – Прим. ред.

12.3.1. Предвзятость представления в обучающих данных *RetroTech*

Давайте сначала проанализируем текущие обучающие данные, чтобы получить представление о ситуации. Каких результатов поиска не хватает в обучающих данных? Где наши знания неполны? Другой способ сказать «предвзятость представления» заключается в том, что есть потенциально релевантные результаты поиска, которые исключены из обучающих данных: слепые пятна, которые мы должны обнаружить и с которыми должны бороться. Как только мы определим эти слепые пятна, сможем лучше их исправить. Это настроит нас на повторное обучение с более надежной моделью.

В листинге 12.5 мы создаем новый набор признаков с именем `explore_feature_set`, в котором у нас есть три простых признака: `long_description_match`, `short_description_match` и `name_match`, сообщающих нам, происходит ли совпадение заданного поля или нет. Они соответствуют признакам, которые наша модель уже изучила. Кроме того, мы добавили признак `has_promotion`. Этот признак получает 1.0, если продукт продается и продвигается через маркетинговые каналы. Мы не исследовали этот признак раньше; возможно, это слепое пятно?

Листинг 12.5. Анализ отсутствующих типов документов

```
def get_latest_explore_feature_set():
    return [
        ltr.generate_query_feature (
            feature_name="long_description_match",
            field_name="long_description",
            constant_score=True),
        ltr.generate_query_feature (
            feature_name="short_description_match",
            field_name="short_description",
            constant_score=True),
        ltr.generate_query_feature (
            feature_name="name_match",
            field_name="name", constant_score=True),
        ltr.generate_query_feature (
            feature_name="has_promotion",
            field_name="has_promotion", value="true")]

def get_logged_transformers_judgments (sessions, features):
    training_data = generate_training_data(sessions)
    logged_judgments = generate_logged_judgments(training_data,
                                                    features, "explore")
```

Признаки, соответствующие полям, которые уже использовались для обучения модели LTR.

Новый признак, который мы исследуем для поиска слепого пятна: рекламные акции.

Строит суждения SDBN из текущих необработанных сеансов.

Регистрирует значения признаков и возвращает суждения SDBN, объединенные со значениями признаков.

```
logged_judgments = logged_judgments \
    [logged_judgments ["query"] == "transformers dvd"]
return logged_judgments
explore_features = get_latest_explore_features()
logged_transformers_judgments = get_logged_transformers_judgments(sessions,
                                                                    explore_features)
```

Изучает свойства текущих обучающих данных «transformers dvd».

```
display(logged_transformers_judgments)
```

Вывод:

doc_id	query	grade	long_desc*_match	short_desc*_match	name_match	has_promotion
97363560449	transformers dvd	0.34	0.0	0.0	1.0	0.0
97361312804	transformers dvd	0.34	0.0	0.0	1.0	0.0
97361312743	transformers dvd	0.34	0.0	0.0	1.0	0.0
97363455349	transformers dvd	0.34	0.0	0.0	1.0	0.0
...						
708056579739	transformers dvd	0.01	1.0	1.0	1.0	0.0
879862003524	transformers dvd	0.01	1.0	1.0	1.0	0.0
93624974918	transformers dvd	0.01	0.0	0.0	1.0	0.0
47875839090	transformers dvd	0.01	1.0	0.0	1.0	0.0

Мы видим некоторые пробелы в знаниях наших обучающих данных в выходных данных листинга 12.5:

- каждый предмет включает совпадение имени;
- нет рекламных акций (has_promotion=0);
- существует диапазон значений long_description_match и short_description_match.

Интуитивно понятно, что если мы хотим расширить наши знания, то должны показать пользователям, ищущим transformers dvd, что-то совершенно выходящее за рамки того, что находится в выходных данных листинга 12.5. Это означало бы показ пользователям продвигаемого предмета и, возможно, предмета без совпадений по имени. Другими словами, нам нужно вывести поиск из его собственной эхо-камеры, явно диверсифицируя то, что мы показываем пользователю, отходя от того, что есть в обучающих данных. Единственный вопрос в том, на какой риск мы готовы пойти, чтобы улучшить наши знания? Вряд ли стоит покрывать результаты поиска случайными продуктами только для того, чтобы расширить наши обучающие данные.

То, что мы сделали до сих пор, не было систематическим: мы проанализировали только один запрос, чтобы увидеть, чего не хватает. Как можно это автоматизировать? Далее мы обсудим один метод автоматизации исследования с помощью инструмента, называемого гауссовым процессом.

12.3.2. За пределами *ad hoc*: вдумчивое исследование с помощью гауссова процесса

Гауссов процесс – это статистическая модель, которая делает прогнозы и обеспечивает распределение вероятностей, фиксирующее достоверность этого прогноза. В этом разделе мы будем использовать гауссов процесс для выбора областей для исследования. Далее в этой главе мы создадим более надежный способ поиска пробелов в наших данных.

Гауссов процесс на примере: исследование новой реки путем использования существующих знаний

Чтобы получить интуитивное представление о гауссовых процессах, давайте используем конкретный пример реального исследования. Работа над этим примером позволит вам более интуитивно думать о том, как мы могли бы математически сделать компромиссы между исследованием и эксплуатацией.

Представьте, что вы ученый, планирующий исследовать малоисследованную реку в дикой местности. Когда вы планируете свою поездку, у вас есть только отдельные наблюдения глубины реки из прошлых экспедиций, чтобы знать, когда безопасно путешествовать. Например, одно наблюдение показывает, что река имеет глубину два метра в апреле; другое время в августе – один метр. Вы хотели бы выбрать дату для своей экспедиции, которая оптимизирует идеальные условия реки (т. е. не сезон муссонов, но и не время засушливого периода). Однако вы также ученый – вы хотели бы проводить наблюдения в еще не наблюдавшееся время года, чтобы расширить свои знания о реке. На рис. 12.5 показаны измерения глубины реки, сделанные в течение года.

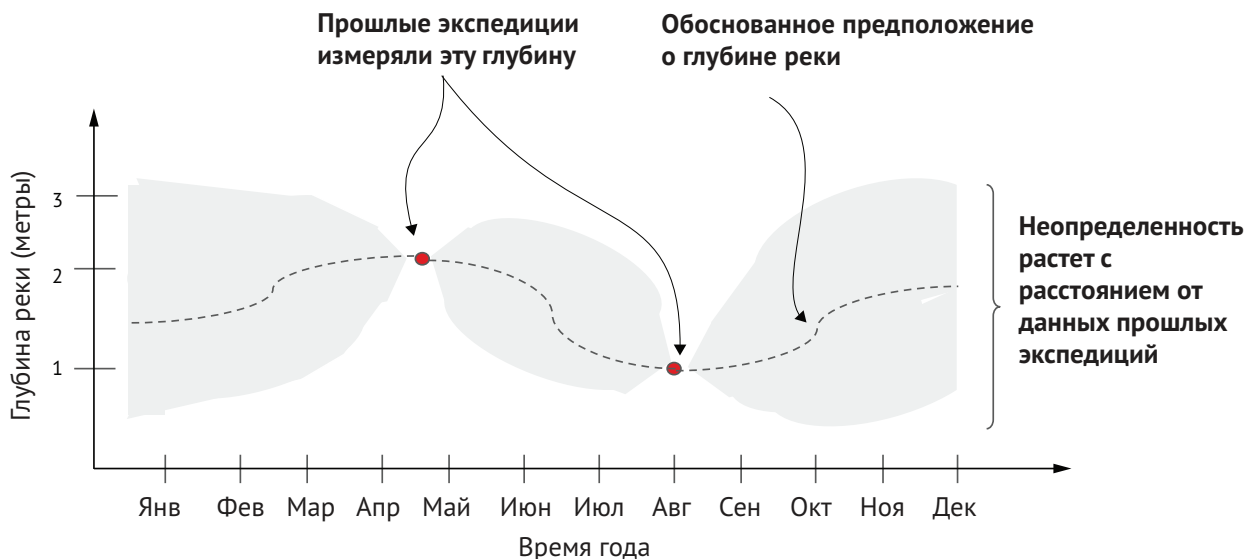


Рис. 12.5. Исследование реки. Неопределенность глубины реки увеличивается по мере удаления от прошлых наблюдений. Как бы мы выбрали время года, которое было бы и безопасным, и максимально увеличивало бы наши знания о глубине реки?

Как бы мы могли выбрать дату для экспедиции? Если бы вы наблюдали глубину реки в два метра 14 апреля, вы бы предположили, что глубина 15

апреля будет очень близка к двум метрам. Путешествие в это время может быть приятным: вы уверены, что река не будет чрезмерно разлита. Однако вы не получите много знаний о реке. А что, если попытаться отойти на несколько месяцев от этого наблюдения, например в январе? Январь будет слишком далек от апреля, чтобы предсказать вероятную глубину реки. Может быть, это опасное время года для путешествия, но мы почти наверняка получим новые знания – возможно, гораздо больше, чем рассчитывали! При таком небольшом количестве данных об этом времени года проводить исследования может быть слишком рискованно.

На рис. 12.5 мы видим обоснованное предположение об уровне реки, основанное на ожидаемой корреляции между соседними датами (15 и 14 апреля должны быть очень близки). Наш уровень уверенности уменьшается по мере того, как мы удаляемся от прямых наблюдений, представленных расширяющейся серой зоной.

На рис. 12.5 показан гауссов процесс. Он математически фиксирует прогнозы вместе с нашей неопределенностью в каждом прогнозе. Как это связано с ранжированием релевантности? Так же как соседние даты имеют схожие уровни реки, схожие результаты поиска будут схожи по своей релевантности. Рассмотрим наши изученные признаки из листинга 12.5. Те, у которых есть сильные совпадения имен для `transformers dvd`, которые не продвигаются и не имеют совпадений краткого или длинного описания, вероятно, будут иметь схожие оценки релевантности – все умеренно релевантные. По мере того как мы отходим от этих хорошо известных примеров – возможно, добавляя продвигаемые предметы, – мы становимся менее уверенными в наших обоснованных догадках. Если мы зайдем слишком далеко, далеко за пределы рамки, например включив в результат поиск без совпадения имен, который продвигается, но имеет сильные совпадения полей краткого или длинного описания, наша неопределенность станет очень высокой. Так же как у ученого, рассматривающего поездку в январе, у нас почти нет возможности сделать хорошее предположение о том, могут ли эти результаты быть релевантными. Показывать эти результаты пользователям может быть слишком рискованно.

Мы можем использовать гауссовы процессы, чтобы сбалансировать использование существующих знаний о релевантности с более рискованным исследованием для получения знаний. Гауссовы процессы используют неполную информацию, чтобы сделать осторожные компромиссы между вероятным качеством и полученными знаниями. Например, мы можем пожертвовать идеальными речными условиями или вероятным релевантным результатом поиска, чтобы узнать больше о речных условиях или о релевантности нового вида результата поиска. Мы можем тщательно выбирать, насколько далеко от известных, безопасных результатов поиска мы хотим зайти, чтобы получить новые знания.

В нашем случае с `transformers dvd` какой результат поиска будет иметь высокий потенциал, будет также релевантным и безопасным для исследования, но также максимально увеличит наши знания? Давайте обучим гауссов процесс и выясним!

Обучение и анализ гауссова процесса

Давайте на практике посмотрим, как работает гауссов процесс. Мы обучим гауссов процесс на запросе трансформеров DVD. Затем мы используем его для генерации лучших кандидатов на разведку. Вы увидите, как мы можем оценить этих кандидатов на разведку, чтобы максимально снизить наш риск и увеличить вероятность получения достоверных знаний. В следующем листинге мы обучаем гауссов процесс с помощью `GaussianProcessRegressor` (он же `gpr`) из `sklearn`. Этот код создает гауссов процесс, который пытается предсказать степень релевантности `grade` как функции признаков `explore_feature_set`, которые мы логировали.

Листинг 12.6. Обучение `GaussianProcessRegressor` на наших обучающих данных

```
from sklearn.gaussian_process import GaussianProcessRegressor

def train_gpr(logged_judgments, feature_names):
    feature_data = logged_judgments[feature_names]
    grades = logged_judgments["grade"]
    gpr = GaussianProcessRegressor()
    gpr.fit(feature_data, grades)
    return gpr
```

Использует признаки, логированные из `explore_feature_set` в листинге 12.5.

Предсказывает степени релевантности.

Создает и обучает модель `gpr`.

После обучения `GaussianProcessRegressor` мы можем использовать его для прогнозирования. Помните, что `GaussianProcessRegressor` не только прогнозирует значение, но и возвращает распределение вероятностей этого прогноза. Это помогает нам оценить достоверность модели.

В листинге 12.7 мы генерируем значения признаков-кандидатов, которые мы хотели бы, возможно, исследовать. В нашем примере исследования реки эти значения соответствуют возможным датам исследования для экспедиции нашего ученого. В нашем случае, поскольку каждый признак может быть либо 0, либо 1, мы рассматриваем каждое возможное значение признака как кандидата.

Листинг 12.7. Прогнозирование набора кандидатов для исследования

```
def calculate_prediction_data(logged_judgments, feature_names):
    index = pandas.MultiIndex.from_product([[0, 1]] * 4,
                                             names=feature_names)
    with_prediction = pandas.DataFrame(index=index).reset_index()
```

Генерирует матрицу кандидатов с возможным значением 0 или 1 для каждого признака, который мы хотим исследовать.

```

Прогнозирует оценку и стандартное отклонение для этих кандидатов на основе распределения вероятности gpr.

gpr = train_gpr(logged_judgments, feature_names)
predictions_with_std = gpr.predict(
    with_prediction[feature_names], return_std=True)
with_prediction["predicted_grade"] = predictions_with_std[0]
with_prediction["predicted_stddev"] = predictions_with_std[1]
Сохраняет прогнозируемую оценку и стандартное отклонение от gpr.

return with_prediction.sort_values("predicted_stddev", ascending=True)

explore_features = get_latest_explore_features()
logged_transformers_judgments = get_logged_transformers_judgments(sessions,
                                                                    explore_features)

feature_names = [f["name"] for f in explore_features]
prediction_data = calculate_prediction_data(logged_transformers_judgments,
                                           feature_names)

display(prediction_data)

```

	long_description_match	short_description_match	name_match	has_promotion	predicted_grade	prediction_stddev
0	0	0	1	0	0.256798	0.000004
1	0	0	1	0	0.014674	0.000005
1	1	1	1	0	0.014864	0.000007
0	1	1	1	0	0.022834	0.000010
1	0	0	1	1	0.018530	0.000010
0	0	0	1	1	0.161596	0.632121
1	1	1	1	1	0.014856	0.632121

В выводе листинга 12.7 мы видим `predicted_grade` – обоснованное предположение от `gpr` о релевантности этого примера. У нас также есть `predict_stddev`, которое захватывает серую полосу на рис. 12.5 – насколько неопределенно предсказание.

Мы отмечаем в выводе листинга 12.7, что стандартное отклонение близко к 0 для первых четырех продуктов с `name_match=1`. Другими словами, `gpr` имеет тенденцию иметь большую достоверность, когда `name_match=1`. После этих наблюдений мы видим, что стандартное отклонение резко увеличивается, поскольку у нас нет большого количества знаний за пределами этих начальных примеров сопоставления имен.

Вывод начинает показывать искажения представления, которые мы интуитивно обнаружили в листинге 12.5. Мы находим огромное количество знаний о важности совпадений имен, но мало знаний о других случаях. Какой случай стоило бы изучить с живыми пользователями, а также минимизировать риск того, что мы покажем пользователям что-то совершенно странное в результатах поиска?

В листинге 12.8 мы генерируем и оцениваем кандидатов на исследование, используя алгоритм, называемый *ожидаемым улучшением*, который предсказывает кандидатов с самым высоким потенциалом роста. Мы рас-

смотрим только основные детали этого алгоритма, поэтому рекомендуем статью «Изучение байесовской оптимизации» Агнихотри и Батры, если вы хотите узнать больше (<https://distill.pub/2020/bayesian-optimization>).

Листинг 12.8. Расчет ожидаемого улучшения для исследования

```
def calculate_expected_improvement(logged_judgments, feature_names,
    theta=0.6):
    data = calculate_prediction_data(logged_judgments, feature_names)
    data["opportunity"] = (data["predicted_grade"] -
        logged_judgments["grade"].mean() - theta)

    data["prob_of_improvement"] = (
        norm.cdf(data["opportunity"] /
            data["predicted_stddev"]))

    data["expected_improvement"] = (
        data["opportunity"] * data["prob_of_improvement"] +
        data["predicted_stddev"] *
        norm.pdf(data["opportunity"] /
            data["predicted_stddev"]))
    return data.sort_values("expected_improvement",
        ascending=False)

improvement_data = calculate_expected_improvement(
    logged_transformers_judgments, feature_names)
display(improvement_data)
```

Будет ли прогнозируемая оценка выше или ниже типичной оценки?

Вероятность, которую мы улучшим по сравнению со средним значением, учитывая степень неопределенности в прогнозе.

Сколько можно выиграть, учитывая вероятность, улучшение и величину улучшения.

Сортировка для отображения лучших кандидатов на исследование.

Вывод:

long_description_match	name_match	short_description_match	has_promotion	opportunity	prob_of_improvement	expected_improvement
0	0	0	1	-0.638497	0.234728	0.121201
0	1	0	1	-0.725962	0.213214	0.110633
0	0	0	0	-0.580755	0.232556	0.107853
1	1	0	1	-0.727500	0.204914	0.101653
0	1	0	0	-0.722661	0.181691	0.078549

Чем выше ожидаемое улучшение, тем выше прогнозируемый рост для кандидата на разведку. Но как алгоритм количественно оценивает этот потенциальный рост? Либо мы знаем с высокой степенью уверенности, что есть рост (стандартное отклонение низкое, а прогнозируемая оценка высокая), либо мы знаем, что есть высокая степень неопределенности, но прогнозируемая оценка все еще достаточно высока, чтобы рискнуть. Мы можем увидеть это в следующем коде из листинга 12.8:

```
data["expected_improvement"] = (
    data["opportunity"] * data["prob_of_improvement"] +
    (data["prediction_stddev"] * norm.pdf(data["opportunity"] /
        data["prediction_stddev"])))
```

Возможность, которая является более чем гарантированной, покрывается этим первым выражением:

```
data["opportunity"] * data["prob_of_improvement"]
```

Между тем неизвестная возможность с большой изменчивостью покрывается этим вторым выражением (после +):

```
data["prediction_stddev"] * norm.pdf(data["opportunity"] /  
                                     data["prediction_stddev"])
```

В первом выражении вы заметите, что возможность (насколько мы ожидаем получить), умноженная на вероятность того, что улучшение произойдет, соответствует ощущению уверенности в хорошем результате. С другой стороны, второе выражение зависит гораздо больше от стандартного отклонения. Чем выше стандартное отклонение *и возможность*, тем больше вероятность, что он будет выбран.

Мы можем откалибровать нашу толерантность к риску с помощью параметра, называемого *theta*: чем выше это значение, тем больше мы предпочитаем кандидатов с более высоким стандартным отклонением. Высокая *theta* приводит к тому, что *opportunity* уменьшается до 0. Это смещает оценку ко второму выражению – неизвестным случаям с более высоким стандартным отклонением.

Если мы установим *theta* слишком высоко, наш *grg* выберет кандидатов для изучения, не учитывая, могут ли они быть полезны пользователю. Если *theta* слишком низка, мы не будем изучать слишком много новых кандидатов и вместо этого будем склоняться к существующим знаниям. Высокая *theta* аналогична ученому, идущему на большой риск (например, исследование в январе) на рис. 12.5, в то время как очень низкая *theta* соответствует путешествию вне риска (например, середина апреля). Поскольку мы используем этот алгоритм для дополнения существующей системы LTR, мы выбрали *theta* 0.6 (немного высоко), чтобы попробовать получить больше знаний.

В выводе листинга 12.8 мы видим, что *grg* подтверждает наш предыдущий анализ *ad hoc*¹: мы должны показывать пользователям продукты с акциями. Эти продукты, скорее всего, дадут больше знаний с возможной высокой выгодой от игры.

Теперь, определив тип продуктов, которые мы должны изучить, давайте соберем продукты из поисковой системы, чтобы показать пользователям. Следующий листинг показывает, как мы можем выбирать продукты для изучения, которые мы позже можем вставить в результаты поиска существующей модели.

¹ Анализ «для данного случая». – Прим. ред.

Листинг 12.9. Выбор продукта для изучения из поисковой системы

```

Исследует в соответствии с предостав-
ленным вектором изучения и выбирает
случайный документ из этой группы.
def explore(query, logged_judgments, features):
    feature_names = [f["name"] for f in features]
    prediction_data = calculate_expected_improvement(logged_judgments,
                                                    feature_names)
    explore_vector = prediction_data.head().iloc[0][feature_names]
    return search_for_explore_candidate(explore_vector, query)

Извлекает призна-
ки лучших кандида-
тов на изучение на
основе expected_
improvement.

Поиск кандидата,
соответствующе-
го критериям.

explore_features = get_latest_explore_features()
logged_judgments = get_logged_transformers_judgments(sessions,
                                                    explore_features)
exploration_upc = explore("transformers dvd", logged_judgments,
                        explore_features)["upc"]
print(exploration_upc)

```

Вывод:

```
826663114164      # Transformers: The Complete Series [25th Anniversary ... ]
```

В листинге 12.9 мы берем лучшего кандидата на исследование – продвигаемые на рынок продукты – и выдаем запрос на выборку документов с этими признаками. Мы опускаем здесь низкоуровневый перевод преобразования кандидата в запрос (функция `explore_query`), но вы можете себе представить, что если `has_promotion=1.0` в кандидате, то мы выдаем запрос на поиск любого продукта с продвижением (`has_promotion=true`) и т. д. для других признаков.

В выводе листинга 12.9 мы видим, что для запроса `transformers dvd` случайно выбранный продвигаемый продукт для исследования – 826663114164. Это соответствует «Transformers: The Complete Series [25th Anniversary Matrix of Leadership Edition] [16 Discs] – DVD». Интересно!

Что нам делать с этим документом? Это сводится к решению по дизайну. Обычный выбор – поместить его на третью позицию результатов, так как это позволяет пользователю сначала видеть наиболее релевантные результаты, но также гарантирует, что результат исследования получит высокую видимость. Обратите внимание на наш документ 826663114164 в третьем слоте (`rank = 2.0`).

doc_id	product_name	sess_id	query	rank	click
93624974918	Transformers: Revenge	0. 100049	transformers dvd	0.0	False
879862003524	Razer - DeathAdder Tran.	100049	transformers dvd	1.0	False
826663114164	Transformers: The Compl.	100049	transformers dvd	2.0	False
708056579739	Nintendo - Transformers.	100049	transformers dvd	3.0	False

Мы смоделировали много похожих сеансов исследования в прилагаемой записной книжке. Поскольку у нас нет реальных живых поль-

зователей, заходящих в наше приложение, код моделирования реализован только в демонстрационных целях, чтобы мы могли интегрировать кандидатов на исследование для активного обучения в нашу автоматизированную систему LTR. В реальной производственной среде вы бы показывали результаты реальным пользователям вместо имитации сеансов пользователей.

Каждый сеанс добавляет случайного кандидата исследования на основе листинга 12.9, имитирует, был ли кликнут добавленный результат исследования, и добавляет его к новому набору сеансов: `sessions_with_exploration`. Напомним, что эти сеансы служат входными данными, которые нам нужны для вычисления данных обучения LTR (суждения на основе SDBN из главы 11, которые генерируются в листинге 12.1).

Наконец, у нас есть данные, необходимые для повторного запуска нашего автоматизированного цикла обучения LTR. Посмотрим, что произойдет с этими примерами, добавленными к нашим данным обучения, и как мы можем вписать это исследование в общий автоматизированный алгоритм LTR.

12.3.3. Изучение результата наших исследований

Мы провели исследование, показав некоторые нестандартные результаты поиска (имитируемым) реальным пользователям. Теперь у нас есть новые сеансы, добавленные к исходным данным сеанса, хранящимся во фрейме данных `sessions_with_exploration`. В этом разделе мы прогоним данные сеанса через наши автоматизированные функции LTR для повторной генерации обучающих данных и обучения модели. Затем запустим эту новую модель в А/В-тесте, чтобы увидеть результаты.

Как вы помните, наши автоматизированные помощники LTR могут повторно генерировать обучающие данные с помощью функции `generate_training_data`. Мы делаем это в листинге 12.10, но на этот раз с нашими дополненными сеансами, которые включают данные исследования.

Листинг 12.10. Повторная генерация суждений SDBN из новых сеансов

```
query = "transformers dvd"
sessions_with_exploration = generate_simulated_exploration_sessions(
    query, sessions, logged_transformers_judgments, explore_features)
training_data_with_exploration = \
    generate_training_data(sessions_with_exploration)
display(training_data_with_exploration.loc["transformers dvd"])
```

Вывод:

doc_id	product_name	click	examined	grade	beta_grade
97360724240	Transformers: Revenge of..	43	44	0.977	0.833333
826663114164	Transformers: The Comple..	42	44	0.954	0.814815
97360722345	Transformers/Transformer.	46	55	0.836	0.738462
97363455349	Transformers Widescree..	731	2113	0.345	0.345266
97361312804	Transformers - Widescree..	726	2109	0.344	0.343558

В выводе листинга 12.10 мы видим, что был включен новый продукт. Обратите особое внимание на добавление 826663114164, «Transformers: The Complete Series [25th Anniversary Matrix of Leadership Edition] [16 Discs] – DVD». Интересно, что у этого фильма `has_promotion=true`, что означает, что он был одним из недавно выбранных кандидатов на исследование из предыдущего раздела:

```
{"upc": "826663114164",
  "name": "Transformers: The Complete Series [25th Anniversary ...] - DVD",
  "manufacturer": "",
  "short_description": "",
  "long_description": "",
  "has_promotion": True}
```

Похоже, пользователей привлекают продвигаемые продукты, поэтому давайте переместим наш признак `has_promotion` из набора признаков изучения в нашу основную модель и переобучим ее, чтобы увидеть эффект. В следующем листинге мы обучаем модель с этим новым признаком, добавленным в смесь, чтобы увидеть эффект.

Листинг 12.11. Перестроение модели с использованием обновленных суждений

```
promotion_feature_set = [
    ltr.generate_fuzzy_query_feature (feature_name="name_fuzzy",
                                     field_name="name"),
    ltr.generate_bigram_query_feature (feature_name="name_bigram",
                                     field_name="name"),
    ltr.generate_bigram_query_feature (feature_name="short_description_bigram",
                                     field_name="short_description"),
    ltr.generate_query_feature (feature_name="has_promotion",
                              field_name="has_promotion",
                              value="true",
                              constant_score=True)]
```

Добавление has_promotion к набору признаков, с помощью которого мы обучаем нашу модель.

```
evaluation = train_and_evaluate_model(sessions_with_exploration,
                                     "ltr_model_variant_3",
                                     feature_set)
```

Оценка для `ltr_model_variant_3`:

```
{"dryer": 0.12737002598513025,      # Before: 0.071
  "blue ray": 0.08461538461538462,   # 0.0
  "headphones": 0.12110565745285455, # 0.065
  "dark of moon": 0.1492224251599605, # 0.258
  "transformers dvd": 0.26947504217124457} # 0.101
```

Ух ты! При сравнении листинга 12.11 с предыдущим выводом из листинга 12.3 мы видим, что добавление продвигаемого продукта к обучающим данным создает значительное улучшение в большинстве

случаев в нашей офлайн-тестовой оценке. Точность `transformers dvd`, в частности, значительно возросла! Если мы выполним поиск `transformers dvd`, мы увидим, что это отражено в наших данных.

Листинг 12.12. Поиск `transformers dvd` с использованием последней модели

```
results = ltr.search_with_model("ltr_model_variant_3",
                               query="transformers dvd",
                               rerank_query="transformers dvd",
                               limit=5)["docs"]
display([doc["name"] for doc in results])
```

Вывод:

```
["Transformers/Transformers: Revenge of the Fallen: Two-Movie Mega Coll...",
 "Transformers: Revenge of the Fallen - Widescreen - DVD",
 "Transformers: Dark of the Moon - Original Soundtrack - CD",
 "Transformers: The Complete Series [25th Anniversary Matrix of Leaders...",
 "Transformers: Dark of the Moon Stealth Force Edition - Nintendo Wii"]
```

Однако мы знаем, что великолепные результаты теста не всегда переносятся в реальный мир. Что произойдет, если мы повторно запустим А/В-тест из листинга 12.4? Если вы помните, мы создали функцию `a_b_test`, которая случайным образом выбирает модель для поиска пользователя. Если результаты содержали предмет, который пользователь тайно хотел купить, покупка, скорее всего, произойдет. Если мы используем эту функцию для повторной имитации А/В-теста, мы увидим, что наша новая модель, похоже, сорвала джекпот!

Листинг 12.13. Повторный запуск А/В-теста на новой модели `ltr_model_variant_3`

```
results = simulate_user_a_b_test(query="transformers dvd",
                                 model_a="ltr_model_variant_1",
                                 model_b="ltr_model_variant_3",
                                 number_of_users=1000)
display(results)
```

Вывод:

```
{"ltr_model_variant_1": 21,
 "ltr_model_variant_3": 145}
```

Теперь мы видим, что новая модель (`ltr_model_variant_3`) значительно превзошла старую модель (`ltr_model_variant_1`) в А/В-тесте. Теперь мы знаем, что не только наше исследование помогло нам найти теоретический пробел в обучающих данных, но и что, когда мы протестировали новую модель в реальном сценарии для нашего целевого запроса (`transformers dvd`), она показала себя значительно лучше, чем старая модель «только для эксплойтов». Хотя в этой главе мы сосредоточились на кон-

кретном запросе, тот же процесс можно применять ко многим запросам и кандидатам на исследование, чтобы продолжить автоматическое совершенствование вашей модели LTR с помощью активного обучения.

Теперь мы внедрили автоматизированную систему LTR, которая не только повторно обучается на основе последних сигналов пользователя, но и использует активное обучение для изучения того, что еще может быть релевантно для реальных пользователей, а затем собирает соответствующие сигналы, измеряя их обратную связь. Этот активный процесс обучения помогает устранять слепые пятна в обучающих данных на постоянной основе.

12.4. Разработка, исследование, сбор, сортировка, повторение: надежный автоматизированный цикл LTR

После того как последние части установлены на своих местах, мы видим, как изучение новых признаков помогает нам преодолеть предвзятость представления. Изучение признаков и исследование обучающих данных идут рука об руку, поскольку мы изучаем наши предвзятости представления, понимая, каких признаков нам не хватает и которые, возможно, необходимо внедрить в поиск. В этой главе мы использовали простой пример с «рекламными акциями», но какие другие, более сложные признаки могут оказаться в слепых пятнах ваших обучающих данных? В этом разделе давайте завершим, дополнив наш автоматизированный алгоритм LTR из главы 11, включив не только обучение модели с предыдущими данными обучения на основе модели кликов, но и этот новый подход активного обучения для исследования за пределами текущего объема данных обучения.

Наш новый алгоритм автоматического исследования LTR можно обобщить в следующих трех основных шагах.

- 1 *Разработка* – использование известных признаков и обучение модели LTR для ранжирования с использованием существующих данных обучения.
- 2 *Исследование* – создание предполагаемых, «исследуемых» признаков для устранения слепых зон данных обучения.
- 3 *Сбор* – с развернутой моделью и обученной моделью `grg` показ результатов поиска исследования, эксплуатации и сбор кликов для построения суждений.

Мы можем суммировать последние три главы, объединив их в листинге 12.14, показанном ниже. Этот листинг объединяет все части вместе (с некоторыми опущенными внутренними предметами). Нашими основными точками принятия решений в этом алгоритме являются признаки, используемые для исследования и эксплуатации. Мы также можем залезть под капот, чтобы изменить выбранную модель кликов, архитектуру модели LTR и нашу толерантность к риску (параметр `theta`).

Листинг 12.14. Подведение итогов полностью автоматизированного алгоритма LTR

```

def train_and_deploy_model(sessions, model_name, feature_set):
    judgments = generate_training_data(sessions)
    train, test = split_training_data(judgments, 0.8)
    train_ranksvm_model(train, model_name, feature_set=feature_set)

def ltr_retraining_loop(latest_sessions, iterations=sys.maxsize,
                       retraining_frequency=60 * 60 * 24):
    exploit_feature_set = get_exploit_feature_set()
    train_and_deploy_model(latest_sessions,
                           "exploit",
                           exploit_feature_set)

    for i in range(0, iterations):
        judgments = generate_training_data(latest_sessions)
        train, test = split_training_data(judgments)
        if i > 0:
            previous_explore_model_name = f"explore_variant_{i-1}"
            exploit_model_evaluation = evaluate_model(test_data=test,
                                                       model_name="exploit", training_data=train)
            explore_model_evaluation = evaluate_model(test_data=test,
                                                       model_name=previous_explore_model_name, training_data=train)
            print(f"Exploit evaluation: {exploit_model_evaluation}")
            print(f"Explore evaluation: {explore_model_evaluation}")
            if is_improvement(explore_model_evaluation,
                              exploit_model_evaluation):
                print("Promoting previous explore model")
                train_and_deploy_model(latest_sessions,
                                       "exploit",
                                       explore_feature_set)

            explore_feature_set = get_latest_explore_feature_set()
            train_and_deploy_model(latest_sessions,
                                   f"explore_variant_{i}",
                                   explore_feature_set)

        wait_for_more_sessions(retraining_frequency)
        latest_sessions = gather_latest_sessions(
            "transformers dvd",
            latest_sessions,
            explore_feature_set)

    ltr_retraining_loop(sessions)

```

Возвращает модель один раз в день.

Обучает модель LTR на известных хороших признаках и текущих данных обучения.

Собирает новые сеансы и повторяет процесс.

Оценивает текущий вариант исследования и продвигает его, если он лучше текущей модели исследования.

Выдвигает гипотезы о новых признаках на предмет исследования слепых зон.

Собирает новые сеансы и повторяет процесс.

Собирает сигналы пользователя, пока не наступит время переобучения модели.

В этом цикле мы фиксируем более автоматизированный процесс LTR. Мы активно изучаем слепые зоны наших обучающих данных, теоретизируя признаки, которые могут в них находиться. Запустив цикл, мы можем наблюдать за его производительностью и решать, когда продвигать признаки «исследований» в полный производственный набор признаков «использования». По мере того как мы удаляем

старые данные о кликах, мы также можем заметить, когда старые признаки больше не имеют значения и когда новые признаки становятся важными из-за новых тенденций и сезонности. Наша реализация использует настроенный вручную набор признаков «использовать и исследовать», чтобы наша команда по релевантности поиска контролировала разработку признаков, но вы, безусловно, можете написать алгоритм для генерации новых признаков или использовать глубокое обучение для обнаружения латентных признаков или использовать какой-либо другой подход на основе существующих свойств контента.

Вместе эти алгоритмы обеспечивают надежный механизм для приближения к идеальному ранжированию, учитывая полный спектр вариантов, которые могут быть показаны пользователям. Они позволяют вам выбирать новые признаки для исследования слепых зон, достигая алгоритма релевантности, который максимизирует то, что пользователи предпочитают видеть в результатах поиска.

Резюме

- Хорошие результаты в офлайн-тесте показывают, что наши признаки могут аппроксимировать данные обучения. Однако это не гарантия успеха. Тест A/B может показать нам ситуации, когда сами данные обучения были вводящими в заблуждение.
- Данные обучения должны отслеживаться на предмет предвзятости и тщательно корректироваться.
- Предвзятость представления – одна из самых пагубных проблем релевантности поиска. Предвзятость представления возникает, когда наши модели не могут узнать, что релевантно из кликов пользователей, потому что результаты никогда не появляются для того, чтобы на них кликнули первый раз.
- Мы можем преодолеть предвзятость представления, сделав автоматизированный процесс LTR активным участником поиска слепых пятен в данных обучения. Модели, которые делают это, участвуют в активном обучении.
- Гауссов процесс – один из способов выбора перспективных возможностей для исследования. Используя набор признаков, мы можем найти то, чего не хватает в обучении, и выбрать новые предметы для показа пользователям на основе того, какие предметы, скорее всего, предоставят наиболее полезные новые точки данных для дальнейшего обучения. Мы можем экспериментировать с различными способами описания данных с помощью признаков, чтобы находить новые и интересные слепые зоны и области исследования.
- Когда мы объединяем использование существующих знаний с исследованием слепых зон, у нас появляется более надежная автоматизированная система LTR – отраженный интеллект, который может автоматически исследовать и использовать признаки с небольшим внутренним обслуживанием.

Часть 4

Передний край поиска

Рост приенения эмбедингов и генеративного ИИ стал благом для области поиска информации. Большие языковые модели (LLM) и другие базовые модели не только предоставляют новые способы понимания и генерации текста, но и служат идеальным дополнением для поисковых систем. Генеративные модели нуждаются в надежных данных в качестве контекста (которые предоставляют поисковые системы), а поисковые системы должны интерпретировать и обобщать данные, которые они ищут (которые предоставляют генеративные модели ИИ).

В части 4 мы рассмотрим границы поиска. Мы изучим, как генеративные модели используются для улучшения поиска и как поиск используется для дополнения генеративных моделей. Мы также рассмотрим будущее на пересечении ИИ и поиска информации.

Глава 13 охватывает семантический поиск по эмбедингам, объясняя, как работают преобразователи и как семантический поиск по плотным векторам можно оптимизировать для эффективности с помощью приближенного ближайшего соседа (ANN) и подходов квантизации.

Глава 14 демонстрирует, как тонко настроить LLM на ваших данных и реализовать извлекающий ответ на вопрос: отвечать на вопросы в запросах явными ответами, извлеченными из результатов поиска.

Глава 15 посвящена обсуждению новых методов поиска на основе ИИ. Мы рассмотрим методы генеративного поиска, продемонстрируем расширенную генерацию поиска (RAG) для динамического суммирования результатов поиска, генерацию синтетических обучающих данных с использованием генеративных моделей и оценку качества генеративной модели. Наконец, мы продемонстрируем мультимодальный поиск (по тексту и изображениям) и гибридный поиск (объединяющий лексический и поиск по плотным векторам) и заглянем в будущее поиска в эпоху генеративного ИИ.

13

Семантический поиск с плотными векторами

В этой главе рассматривается:

- семантический поиск с использованием эмбеддингов из LLM;
- введение в трансформеры и их влияние на представление и поиск текста;
- создание автозаполнения с использованием моделей трансформеров;
- использование поиска ANN и векторной квантизации для ускорения поиска по плотным векторам;
- семантический поиск с би-кодировщиками и кросс-кодировщиками.

В этой главе мы начнем наше путешествие в поиск по плотным векторам, где гиперконтекстные векторы, генерируемые большими языковыми моделями (LLM), приводят к значительным улучшениям в интерпретации запросов, документов и результатов поиска. Генеративные LLM (например, ChatGPT от OpenAI и многие другие коммерческие и открытые альтернативы) также могут использовать эти векторы для генерации нового контента, включая расширение запросов, данные обучения поиска и суммирование результатов поиска, как мы рассмотрим далее в следующих главах.

Современное состояние LLM меняется ежемесячно (а иногда и почти ежедневно), но даже лучшие модели общего назначения в насто-

ящее время могут быть превзойдены в определенных задачах путем тонкой настройки других меньших моделей для этих задач. В следующих нескольких главах мы обсудим понятия, лежащие в основе LLM, и то, как лучше всего использовать их в вашем поисковом приложении. В этой главе мы познакомимся с трансформерами¹ и обсудим, как использовать их для семантического поиска с плотными векторами. Мы рассмотрим тонкую настройку LLM для ответов на вопросы в главе 14 и использование LLM и других базовых моделей для генеративного поиска в главе 15.

Наша история начинается с того, что вы узнали в разделе 2.5: что мы можем представлять контекст как числовые векторы, и мы можем сравнивать эти векторы, чтобы увидеть, какие из них ближе, используя метрику сходства. В главе 2 мы продемонстрировали концепцию *поиска по плотным векторам*, метод, известный как поиск по плотным векторам, но наши примеры были простыми и надуманными (поиск по выдуманному атрибуту еды). В этой главе мы поставим вопросы «Как мы можем преобразовать реальный неструктурированный текст в многомерное плотное векторное пространство, которое пытается моделировать фактическое значение текстового представления?» и «Как мы можем использовать это представление знаний для продвинутых поисковых приложений?».

13.1. Представление смысла посредством эмбедингов

Мы собираемся использовать языковой перевод в качестве примера, чтобы понять, что мы подразумеваем под эмбедингами плотных векторов. Возьмите следующие два предложения: «Hello to you!» (английский) и «Barev Dzes» (армянский). Эти два выражения имеют примерно одинаковое значение: каждое из них является приветствием с некоторым оттенком формальности.

С точки зрения вычислений, чтобы успешно ответить на приветствие «Hello to you!», машина должна как понять смысл подсказки, так и понять все возможные идеальные ответы в одном и том же векторном пространстве. Когда ответ определен, машина должна затем выразить его человеку, сгенерировав метку из векторного представления ответа.

Это векторное представление смысла называется эмбедингом. Эмбединги используются взаимозаменяемо между задачами обработки естественного языка (NLP) и могут быть дополнительно сформированы

¹ Трансформер (англ. *transformer*) в программировании – это архитектура нейронной сети, используемая для выполнения задач машинного обучения. Такая архитектура позволяет модели обрабатывать входные данные параллельно, что делает ее высокоэффективной для задач, связанных с последовательными данными. Некоторые области применения трансформеров: распознавание речи, компьютерное зрение, системы рекомендаций (персонализированные рекомендации на основе предпочтений пользователя), генерация текста и музыки. – *Прим. ред.*

для соответствия конкретным вариантам использования. Мы сгенерировали эмбединги из LLM (all-mpnet-base-v2) в главе 9, но упустили большую часть деталей о том, как они работают. В этой главе мы познакомим вас с методами и инструментами для получения эмбедингов из текста и будем использовать их для значительного улучшения интерпретации запросов и документов в нашей поисковой системе.

Обработка естественного языка

Обработка естественного языка, NLP (Natural language processing), – это набор методов и инструментов, которые преобразуют неструктурированный текст в данные, пригодные для машинного процессинга. Область NLP довольно обширна и включает в себя множество областей исследований и типов проблем (задач NLP), которые необходимо решить. Полный список проблемных областей поддерживается на сайте NLP-Progress (<https://nlpprogress.com>).

Мы сосредоточимся конкретно на применении NLP для поиска информации, что является важным требованием поиска на основе ИИ.

Сразу стоит отметить один важный момент: за двумя короткими английскими и армянскими приветствиями, которые мы упомянули, скрываются глубокие культурные нюансы. Каждое из них несет в себе богатую историю, и их изучение, таким образом, несет в себе контекст этих историй. Так было с семантическими графами знаний, которые мы исследовали в главе 5, но они использовали в качестве модели только контекст документов в поисковой системе. Трансформаторы обычно обучаются на гораздо больших объемах текста, привнося значительно больше этого тонкого контекста из внешних источников.

Мы можем использовать человеческий мозг в качестве аналогии того, как модели трансформеров учатся представлять значение. Как вы, будучи младенцем, ребенком, подростком и старше, узнавали значение слов? Вам рассказывали, и вы потребляли знания и их представление. Люди, которые учили вас, уже имели эти знания и силу их выражать. Помимо того, что кто-то указывал на кошку и говорил вам «кошечка», вы также смотрели фильмы и видео, а затем переходили к литературе и учебным материалам. Вы читали книги, блоги, периодические издания и письма. Благодаря всему вашему опыту вы встраивали эти знания в свой мозг, создавая плотное представление понятий и того, как они соотносятся друг с другом, что позволяло вам рассуждать о них.

Можем ли мы передать машинам тот же контент, из которого мы получили эту силу языка, а затем ожидать, что они поймут и разумно ответят, когда их спросят? Держитесь за свои шляпы!¹

¹ Автор имеет в виду известную музыкальную комедию-вестерн 1940 года «Hold On To Your Hats». В данном контексте слыш этой фразы означает: «Держитесь! Приключения только начинаются!» – *Прим. ред.*

13.2. Поиск с использованием плотных векторов

Чтобы понять, когда следует использовать плотные векторы для поиска вместо разреженных векторов, необходимо понимать, как обрабатывать и связывать текст. В этом разделе кратко рассматривается, как работает поиск разреженных векторов по сравнению с поиском плотных векторов. Мы также представим *поиск ближайших соседей* как один из типов сходства, используемый для поиска плотных векторов, по сравнению с BM25 (наиболее распространенная функция сходства, используемая для поиска разреженных векторов).

Поиск ближайших соседей на основе векторов

Также известный как KNN (*k*-ближайший сосед) поиск ближайших соседей на основе векторов представляет собой проблемное пространство индексации числовых векторов однородной размерности в структуру данных и поиска в этой структуре данных с помощью вектора запроса для ближайших *k*-связанных векторов. Мы упоминали в главе 3, что существует множество мер сходства для сравнения числовых векторов: косинусное сходство, скалярное произведение, евклидово расстояние и т. д. Мы будем использовать косинусное сходство (реализованное как скалярное произведение векторов, нормированных по единице) в этой главе для сравнения сходства векторов.

13.2.1. Краткая информация о разреженных векторах

Поиск по разреженным векторам¹ обычно реализуется с использованием инвертированного индекса. Инвертированный индекс похож на то, что вы найдете в конце любого учебника, – список терминов, которые ссылаются на их локацию в исходном контенте. Чтобы эффективно находить текст, мы структурируем информацию в индексе, обрабатывая и нормируя токены в словарь со ссылками на публикации (идентификаторы документов и позиции, в которых они встречаются). Результирующая структура данных представляет собой разреженное векторное представление, которое позволяет быстро находить эти токены.

Во время поиска мы токенизируем и нормируем термины запроса и, используя инвертированный индекс, сопоставляем совпадения документов для извлечения. Затем мы применяем формулу BM25 для оценки документов и ранжирования их по сходству, как было рассмотрено в разделе 3.2.

Применение оценок для каждого термина запроса и признака документа дает нам быстрый и релевантный поиск, но эта модель страдает

¹ Поиск по разреженным векторам позволяет использовать сопоставление по ключевым словам во всем контенте. Текст преобразуется в векторы, где подсчитывается, сколько раз каждое уникальное слово встречается в запросе и предложениях. Такие векторы в основном состоят из нулей, так как вероятность того, что любое предложение содержит все слова из словаря, крайне мала. – *Прим. ред.*

от модели релевантности «зависимость от термина запроса», в которой термины (и нормированные формы) извлекаются и ранжируются. Проблема в том, что она использует наличие (и количество) строк терминов запроса для поиска и ранжирования вместо значения, представленного этими строками. Таким образом, оценки релевантности полезны только в относительном смысле, чтобы сообщить вам, какие документы лучше всего соответствуют запросу, но не для измерения того, были ли какие-либо документы объективно хорошими соответствиями. Плотные векторные подходы, как мы увидим, могут обеспечить более глобальное чувство релевантности, которое также имеет свои плюсы в ответах на запросы.

13.2.2. Система поиска по плотным векторам¹

Мы хотим уловить смысл контента при обработке документов и хотим извлекать и ранжировать на основе смысла и намерения запроса при поиске. Имея в виду эту цель, мы обрабатываем документы для генерации эмбедингов, а затем сохраняем эти эмбединги в поисковом индексе. Во время поиска мы обрабатываем запросы для получения эмбедингов и используем их для поиска в индексированных эмбедингах документов. На рис. 13.1 показана упрощенная схема этого процесса, которую мы рассмотрим подробнее в разделе 13.4.

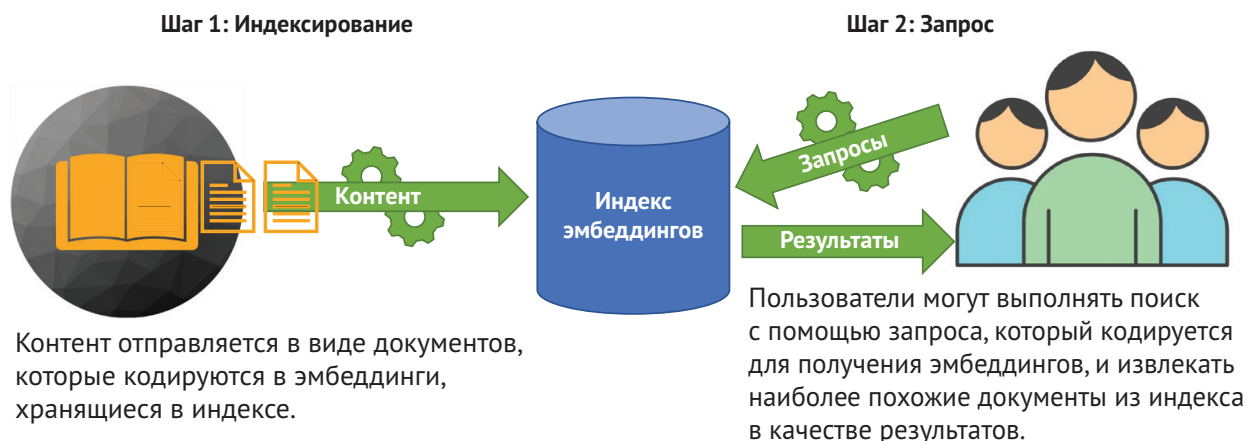


Рис. 13.1. Создание и поиск индекса эмбедингов. Контент обрабатывается и добавляется в индекс слева, и пользователь запрашивает индекс для получения результатов

Эмбединги для документов и запросов существуют в одном и том же векторном пространстве. Это очень важно. Если вы сопоставляете документы с одним векторным пространством, а запросы – с другим, вы буде-

¹ *Dense vector retrieval* в программировании – это поиск по плотным векторам. Принцип работы: каждый из документов в корпусе с помощью нейросети кодируется в плотный вектор. Эти низкоразмерные плотные представления используются для построения векторного индекса. Во время поиска входной запрос встраивается в то же скрытое пространство, и вычисляется схожесть векторов между запросом и документами. Такой поиск может возвращать наиболее похожие результаты на основе векторного расстояния даже при отсутствии точных текстовых совпадений. Эта возможность позволяет получить более тонкие и учитывающие контекст результаты поиска, часто улавливая взаимосвязи между понятиями, которые могут быть упущены при использовании подходов, основанных на ключевых словах. – Прим. ред.

те сопоставлять яблоки с апельсинами. Эмбединги должны принадлежать одному и тому же пространству, чтобы это работало эффективно.

Но что такое эмбединг на самом деле и как мы его ищем? Что ж, эмбединг – это вектор некоторого заданного числа измерений, представляющий информацию. Эта информация может быть запросом, документом, словом, предложением, изображением или видео или любым другим типом информации.

Область и чанкинг эмбединга

Одной из важных инженерных задач при работе с эмбедингами является определение правильного уровня гранулярности для эмбединга. Он может быть создан для представления одного слова, предложения, абзаца или гораздо большего документа.

При создании эмбедингов часто бывает полезно разбить большие документы на разделы и создать отдельный эмбединг для каждого раздела – процесс, известный как *чанкинг* (фрагментация). Вы можете разбить свой контент на части по предложению, абзацу или другим понятийным границам, и вы даже можете создавать перекрывающиеся части, чтобы гарантировать, что процесс расщепления документа не разрушит соответствующий контекст между фрагментами.

Если ваша поисковая система поддерживает многозначные векторные поля, вы можете индексировать множество эмбедингов в один документ и делать сопоставление на основе любого из его эмбедингов. В качестве альтернативы вы можете индексировать отдельный документ для каждого чанка, каждый с одним эмбедингом, а затем сохранять исходный идентификатор документа в качестве поля, которое будет возвращаться при сопоставлении индексированного документа фрагмента.

Сложно полностью представить очень большие фрагменты эмбедингом, так же как сложно очень маленьким чанкам содержать полный контекст, необходимый для эмбединга, поэтому определение правильной детализации чанкинга для вашего приложения может быть важным фактором улучшения отзыва.

Поскольку эмбединги представлены в виде векторов, мы можем использовать косинусное сходство (которое было подробно рассмотрено в главах 2–3) или другое похожее измерение расстояния, чтобы сравнить два вектора эмбединга друг с другом и получить оценку сходства. Это позволяет нам сравнивать вектор запроса с векторами всех документов в контенте, который мы хотим найти. Векторы документов, которые наиболее похожи на вектор запроса, называются *ближайшими соседями*. Рисунок 13.2 иллюстрирует это с помощью трех двумерных векторов. Косинусное сходство между векторами, показанными на рис. 13.2, упорядоченными по наибольшему сходству, выглядит следующим образом:

- $\cos(b, c) = 0.9762$,
- $\cos(a, b) = 0.7962$,
- $\cos(a, c) = 0.6459$.

Ясно и визуально, и математически, что $\mathbf{b}$ и $\mathbf{c}$ находятся ближе всего друг к другу, поэтому мы говорим, что $\mathbf{b}$ и $\mathbf{c}$ являются наиболее похожими из трех векторов.

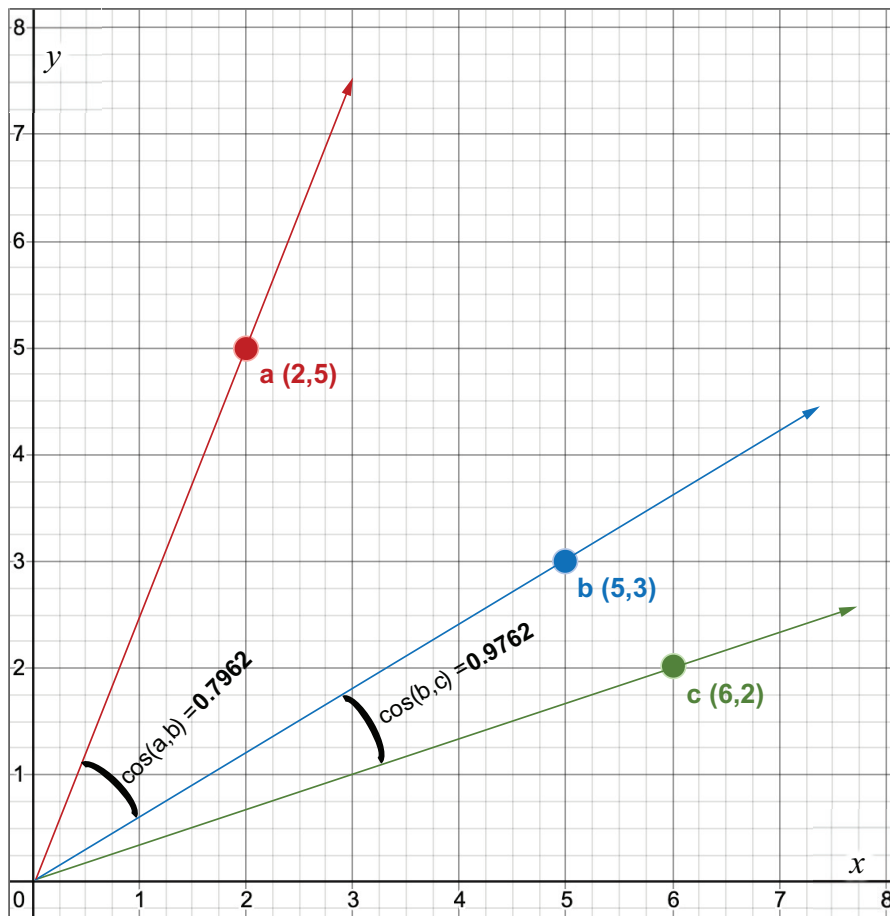


Рис. 13.2. Три вектора ($\mathbf{a}$, $\mathbf{b}$ и $\mathbf{c}$), нанесенные на декартову плоскость. Сходства между $\mathbf{a}$ и $\mathbf{b}$, а также между $\mathbf{b}$ и $\mathbf{c}$, проиллюстрированы с помощью функции $\cos$

Мы можем легко применить косинусное сходство к векторам любой длины. В трехмерном пространстве мы сравниваем векторы с тремя признаками $[x, y, z]$. В пространстве плотного векторного эмбединга мы можем использовать векторы с сотнями или тысячами измерений. Но независимо от количества измерений формула та же самая, как показано на рис. 13.3.

$$\text{Векторное сходство } (\mathbf{a}, \mathbf{b}) : \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| \times |\mathbf{b}|}$$

Рис. 13.3. Формула для косинусного сходства двух векторов

Смотрите раздел. 3.1 для обзора использования этого расчета косинусного сходства для оценки сходства между векторами. Там мы рассмотрели примеры расчета как косинусного сходства между векторами, так и скалярного произведения векторов. Однако из формулы для косинусного сходства (рис. 13.3) вы можете видеть, что косинус равен скалярному произведению $(\mathbf{a} \cdot \mathbf{b})$, деленному на произведение длин

векторов ($|a| \times |b|$). Это означает, что, если мы можем нормировать признаки в векторах a и b так, чтобы каждая из их длин была равна 1 (процесс, называемый *единичным нормированием*), косинусное сходство и скалярное произведение будут равны друг другу:

$$\begin{aligned} 1 &= |a| = |b| \\ \cos(a, b) &= (a \cdot b) / |a| \times |b| \\ \cos(a, b) &= (a \cdot b) / (1 \times 1) \\ \cos(a, b) &= a \cdot b \end{aligned}$$

Когда вектор нормирован таким образом, что его длина равна 1, он известен как *единичный вектор*. Но зачем нам беспокоиться о нормировании векторов таким образом? Что ж, оказывается, что вычисление скалярного произведения гораздо эффективнее вычисления косинусного сходства, потому что нет необходимости делить скалярное произведение на длины каждого вектора (что требует использования теоремы Пифагора для вычисления квадратного корня из суммы квадратов признаков каждого вектора). Поскольку вычисление косинуса часто является самой затратной частью поиска при оценке большого количества документов, единичные векторы во время индексации и выполнения скалярного произведения индексированных векторов с единичным вектором запроса во время поиска могут существенно ускорить поиск, обеспечивая тот же результат:

```
vector_a = [5.0, 3.0]
vector_b = [6.0, 2.0]
```

```
unit_vector_a
= unit_normalize(vector_a)
= unit_normalize([5.0, 3.0])
= [5.0 / sqrt(5.0^2 + 3.0^2), 3.0 / sqrt(5.0^2 + 3.0^2)]
= [0.8575, 0.5145]
```

```
unit_vector_b
= unit_normalize(vector_b)
= unit_normalize([6.0, 2.0])
= [6.0 / sqrt(6.0^2 + 2.0^2), 2.0 / sqrt(6.0^2 + 2.0^2)]
= [0.9487, 0.3162]
```

```
cos(vector_a, vector_b)
= cos([5.0, 3.0], [6.0, 2.0])
= (5.0 x 6.0 + 3.0 x 2.0) /
  (sqrt(5.0^2 + 3.0^2) x sqrt(6.0^2 + 2.0^2))
= 0.9762
```

```
dot_product(unit_vector_a, unit_vector_b)
= dot_product([0.8575, 0.5145], [0.9487, 0.3162])
= (0.8575 x 0.9487) + (0.5145 x 0.3162)
= 0.9762
```

```
cos(vector_a, vector_b) = dot_product(unit_vector_a,
unit_vector_b) = 0.9762.
```

Нормирование векторов a и b к единичному вектору. Все индексированные векторы проходят это нормирование один раз, перед индексацией.

Полный расчет косинусного сходства. Обратите внимание, что знаменатель извлекает квадратный корень из суммы квадратов для каждого вектора.

Вычисление скалярного произведения для единичных векторов. Обратите внимание на отсутствие знаменателя и гораздо более простую сумму умноженных весов признаков.

Хотя мы концептуально все еще выполняем косинусное сходство (из-за единичных векторов), использование скалярного произведения позволяет нам выполнять существенно более быстрые вычисления во время запроса. Поскольку эта оптимизация возможна, с точки зрения производительности не очень хорошо выполнять полное вычисление косинусного сходства в производстве. Хотя есть веские причины, специфичные для конкретного случая использования, по которым вы захотите вычислить косинус, а не скалярное произведение, например для игнорирования длины векторов (см. раздел 3.1.4 для освежения знаний), вы практически всегда будете по крайней мере реализовывать косинусное сходство с использованием единичных векторов и вычисления скалярного произведения по соображениям производительности. Чтобы закрепить передовой опыт, мы будем последовательно использовать этот шаблон во всех оставшихся листингах кода, когда реализуется косинусное сходство.

Оптимизация производительности и затратность вычислений векторного поиска

Выполнение вычислений векторного сходства может быть медленным и затратным в вычислительном отношении при масштабировании, поэтому важно понимать, как сделать правильные компромиссы для оптимизации производительности и затратности.

Поскольку скалярное произведение вычисляется значительно быстрее, чем косинусное сходство, мы рекомендуем всегда реализовывать вычисление косинусного сходства путем индексации векторов, нормированных на единицу, а затем выполнять вычисления скалярного произведения между векторами документа и вектором запроса, нормированным на единицу, во время поиска. Кроме того, часто можно сэкономить значительную часть памяти и существенно улучшить время поиска, используя другие методы оптимизации:

- *использование приближенных методов поиска ближайших соседей (ANN) для быстрой фильтрации высших N результатов для ранжирования вместо обработки всех документов (рассматривается в разделе 13.5.3);*
- *квантизацию (сжатие) векторов для уменьшения количества бит, используемых для представления каждой функции в векторном виде (рассматривается в разделе 13.7);*
- *использование обучения представлению Matryoshka (MRL) только для индексации или поиска по ключевым частям ваших эмбедингов, при этом сохраняя большую часть вашего отзыва (рассматривается в разделе 13.7);*
- *чрезмерный запрос ограниченного количества оптимизированных результатов поиска с использованием менее затратного алгоритма поиска или метрики сходства, а затем реранкинг высших N результатов с использованием более дорогой метрики сходства (рассматривается в разделах 13.5.3 и 13.7).*

Имея возможность выполнять поиск по плотным векторам и ближайшим соседям, нашим следующим важным шагом будет поиск способа генерации этих загадочных эмбедингов.

13.3. Получение текстовых эмбедингов с помощью трансформер-кодировщика

В этом разделе мы рассмотрим трансформеры и то, как они работают для представления смысла. Мы также обсудим, как они используются для кодирования этого смысла в эмбединги.

13.3.1. Что такое трансформер?

Трансформеры – это класс архитектур глубоких нейронных сетей, которые оптимизированы для кодирования смысла в виде эмбедингов, а также для обратного декодирования смысла из эмбедингов в текст. Текстовые трансформеры делают это, сначала представляя метки терминов как плотные векторы, используя их окружающий контекст в предложении (часть кодирования), а затем используя выходную модель для перевода векторов в различные текстовые представления (часть декодирования).

Одной из прекрасных особенностей этого подхода является разделение задач между кодированием и декодированием. Мы воспользуемся этой возможностью и используем механизм кодирования только для получения эмбедингов, которые затем можно использовать в качестве семантического представления смысла независимо от каких-либо шагов декодирования.

Представление смысла

Вспомните пример приветствий на английском и армянском языках из введения к разделу 13.1. Используя специализированный трансформер и набор данных для перевода с английского на армянский язык, можно было бы обучить модель кодированию двух фраз «Hello to you!» и «Barev Dzes» в почти идентичные плотные векторы. Затем эти векторы можно было бы декодировать обратно в текст для перевода или воссоздания чего-то более близкого к исходному тексту.

Давайте начнем наше путешествие в страну трансформеров с понимания того, как обучаются модели трансформер-кодировщиков и чему они в конечном итоге учатся. Чтобы понять мотивы и механизмы, лежащие в основе трансформеров, важно знать некоторую историю базовых понятий.

На дворе 1953 год. Вы находитесь в классе с 20 другими учениками, каждый из которых сидит за своей партой. На вашем столе лежит карандаш и лист бумаги с предложением Q: I went to the _____ and bought some vegetables. Вы уже знаете, что делать, и пишете «store» (магазин) на месте пробела. Вы смотрите на одноклассника за соседней партой, а он пишет «market» (рынок). Звенит звонок, и ответы подсчитываются. Самый распространенный ответ – «store», есть несколько с «mar-

ket» и несколько с «gросег» (бакалейщик). Это тест Клоуза. Он предназначен для проверки понимания прочитанного.

Теперь вы переноситесь в 1995 год. Вы сидите в другом классе со студентами, которые проходят другой тест. На этот раз на вашем листе бумаги очень длинный абзац. Он выглядит примерно в 60 слов и довольно сложный:

Тонкая местность в излучине реки, в которой мы оказались, находилась в двадцати милях от моря. Мое первое наиболее яркое и широкое впечатление о единства всего, как мне кажется, пришло в этот памятный дождливый день ближе к вечеру именно тут. В тот момент, когда я понял, что это унылое место, заросшее крапивой, было кладбищем.

После абзаца идет вопрос с пробелом для ответа: «Как далеко от моря находится кладбище?» Ответ: «_____». Вы написали на месте пробела: «В 20 милях». Поздравляю! Вы только что ответили на один из дюжины вопросов в тесте на понимание прочитанного – Regents English. В частности, этот вопрос проверял ваше внимание.

Эти два теста являются основополагающими для понимания того, как мы измеряем понимание письменной речи. Чтобы пройти эти тесты, вы должны читать, читать, читать и еще раз читать. Фактически к тому времени, когда большинство людей сдают эти тесты в школе, они уже практиковали чтение около 14 лет и накопили огромное количество контекстных знаний. Эти теории формируют основу для LLM – моделей NLP, обученных на большом количестве текста (например, на полном наборе данных Common Crawl в интернете).

Крупные прорывы в области NLP достигли кульминации в статье 2018 года исследователей из Google (Джейкоб Девлин и др.) под названием «BERT: Предобучение глубоких двунаправленных трансформеров для понимания языка» («BERT: Pre-training of Deep Bidirectional transformers for Language Understanding»), в которой использовался тест Cloze и механизмы внимания с трансформерами для достижения самых современных показателей на многих тестах понимания языка (<https://arxiv.org/pdf/1810.04805>).

Модель BERT, в частности, выполняет самообучение, предлагая тесты Cloze самому себе. Стиль обучения – «самоконтролируемый», что означает, что это контролируемое обучение, оформленное как неконтролируемая задача. Это идеальный вариант, поскольку не требует ручной предварительной маркировки данных для первоначальной предварительной подготовки модели. Вы можете дать ей любой текст, и она сама сделает тесты. В контексте обучения тест Cloze известен как *маскированное моделирование языка*¹. Модель начинает с более про-

¹ Маскированное моделирование языка (MLM) – это тип машинного обучения, используемый в обработке естественного языка (NLP). Принцип работы MLM заключается в маскировании части ввода и обучении модели предсказывать пропущенные токены – по сути, реконструировать немаскированный ввод. Маскированные языковые модели используются для предсказания следующего слова в предложении, идентификации объектов в предложении, классификации настроений и пр. – Прим. ред.

стого эмбединга (например, с использованием известных библиотек word2vec или GloVe для каждого слова в словаре) и случайным образом удаляет 15 % токенов в предложении для теста. Затем модель оптимизирует функцию потерь, что приведет к более высокому показателю успешности теста Cloze. Кроме того, в процессе обучения она использует окружающие токены и контексты (внимание). Учитывая вектор в одном примере обучения, полученный обученный выходной вектор представляет собой эмбединга, который содержит глубоко изученное представление слова и окружающие контексты.

Мы призываем вас узнать больше о трансформерах и BERT, если вам интересно, прочитав вышеупомянутую статью. Однако все, что вам нужно понять на данный момент, – это как получить эмбединги из кодировщика BERT. Схема процесса показана на рис. 13.4.

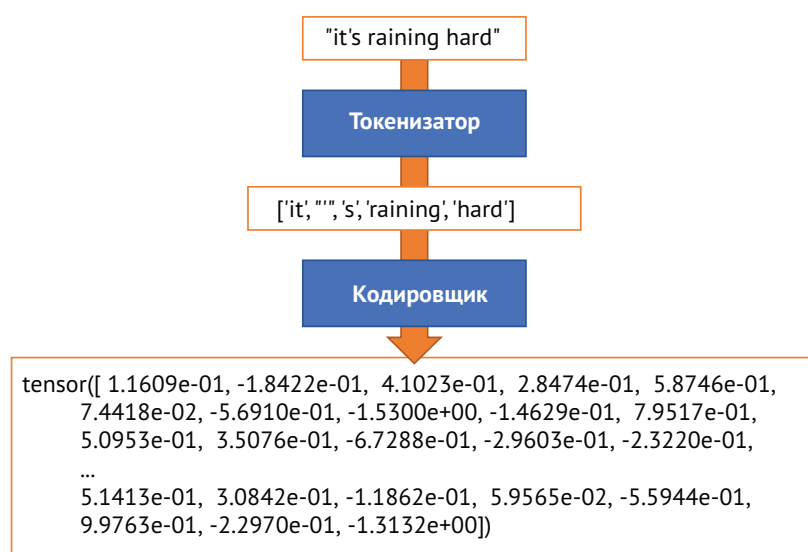


Рис. 13.4. Трансформер-кодировщик

На рис. 13.4 мы обрабатываем текст, сначала пропуская его через токенизатор. Токенизатор разбивает текст на *части слов*, которые являются предопределенными частями слов, представленных в словаре. Этот словарь устанавливается для модели до ее обучения. Например, термин «It's» будет разделен на три части слов во время токенизации: «it», «'» и «s». Словарь, используемый в статье BERT, включал 30 000 частей слов. BERT также использует специальные части слов для обозначения начала и конца предложений: [CLS] и [SEP] соответственно. После токенизации поток токенов передается в модель BERT для кодирования. Затем процесс кодирования выводит тензор, который представляет собой массив векторов (один вектор для каждого токена).

13.3.2. Открытые предобученные модели трансформеров

Хотя трансформеры позволяют создавать самые современные языковые модели, наличие знаний и ресурсов для их создания с нуля может стать для многих серьезным препятствием. Одним из очень

важных аспектов работы с трансформерами является большое сообщество и открытые наборы инструментов, которые позволяют любому инженеру быстро приступить к работе с этой технологией. Все, что нужно, – это некоторые знания Python и подключение к интернету.

Модели, которые обучаются этим процессом с нуля, занимают большой объем памяти, который варьируется от сотен мегабайт до сотен гигабайт, часто требуя аналогичных объемов памяти GPU (VRAM) для их быстрого запуска. Само обучение также требует большого количества затратной вычислительной мощности и большого времени, поэтому возможность использовать уже существующие модели в качестве отправной точки дает значительное преимущество. Мы воспользуемся этим преимуществом в следующем разделе, когда начнем применять одну из этих моделей для поиска.

13.4. Применение трансформеров для поиска

В этом разделе мы создадим высокоточное автозаполнение на естественном языке для поиска, которое будет рекомендовать более точные и иным образом связанные ключевые слова на основе префикса терминов. Мы сделаем это, сначала пропустив наш текст корпуса через трансформер, чтобы получить индекс эмбедингов. Затем мы будем использовать этот трансформер во время запроса, чтобы получить эмбединг запроса и выполнить поиск индекса эмбединга для k -ближайших документов с наиболее похожими эмбедингами. Рисунок 13.5 представляет собой архитектурную диаграмму, демонстрирующую шаги этого процесса.

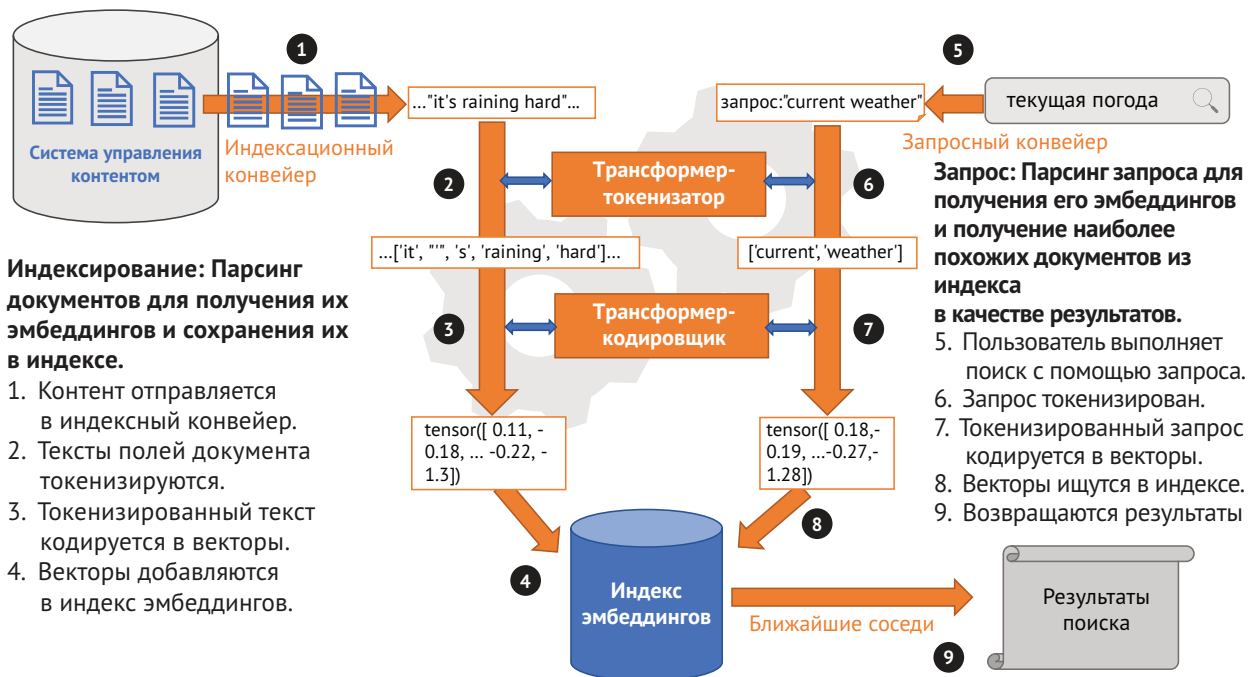


Рис. 13.5. Концептуальная архитектура для сквозного поиска с использованием векторов, созданных трансформером-кодировщиком

У нас есть источник контента, индекс ближайшего соседа, способ извлечения векторов из трансформера и формула подобия. Теперь мы можем построить конвейеры всех этих частей для обработки и индексации контента, а затем для извлечения и ранжирования документов с помощью запроса.

13.4.1. Использование набора данных *Stack Exchange outdoors*

В главе 5 мы представили несколько наборов данных из Stack Exchange¹. Мы решили использовать здесь другой, набор данных Stack Exchange outdoors², по очень важной причине: словарь и контексты в области вопросов и ответов outdoor уже имеют хорошее покрытие в моделях трансформеров, которые мы будем использовать. В частности, Wikipedia используется при обучении многих моделей трансформера, и в Wikipedia есть раздел, специально посвященный контенту outdoor (<https://en.wikipedia.org/wiki/outdoor>).

ПРИМЕЧАНИЕ Примеры контента outdoor, используемые в главах 13–15, а также ссылки на наборы данных Stack Exchange в других главах, лицензированы Stack Exchange по лицензии CC-by-SA 4.0: <https://creativecommons.org/licenses/by-sa/4.0>.

В следующем листинге показано создание коллекции outdoors и последующее индексирование данных вопросов и ответов outdoor.

Листинг 13.1. Индексирование набора данных outdoors

```
outdoors_collection = engine.create_collection("outdoors")
outdoors_dataframe = load_outdoors_data("data/outdoors/posts.csv")
outdoors_collection.write(outdoors_dataframe)
```

Это схема для коллекции outdoors, созданной в листинге 13.1:

```
|-- id: integer (nullable = true)
|-- accepted_answer_id: integer (nullable = true)
|-- parent_id: integer (nullable = true)
|-- creation_date: timestamp (nullable = true)
|-- score: integer (nullable = true)
|-- view_count: integer (nullable = false)
|-- body: string (nullable = true)
|-- owner_user_id: string (nullable = true)
|-- title: string (nullable = true)
|-- tags: array (nullable = true)
```

¹ Stack Exchange – сеть веб-сайтов для работы с вопросами и ответами в различных областях. – Прим. ред.

² The Great outdoors Stack Exchange – это сайт вопросов и ответов для людей, которые любят проводить время на природе, наслаждаясь дикой местностью, и хотят узнать о необходимых навыках и оборудовании. Сайт доступен по адресу *outdoors.stackexchange.com*. – Прим. ред.

```
| |-- element: string (containsNull = true)
|-- answer_count: integer (nullable = true)
|-- post_type: string (nullable = true)
|-- url: string (nullable = true)
```

Индексированный набор данных содержит документы, представляющие как вопросы, так и ответы, с ответами, связанными с их исходными вопросами через поле `parent_id`. Каждый документ содержит поле `post_type`, чтобы различать, содержит ли он вопрос или ответ.

В следующем листинге показан пост с вопросом об узлах для скалолазания и связанные с ним ответы.

Листинг 13.2. Изучение данных поста для вопроса о climbing knots

```
[{"id": "18825",
  "accepted_answer_id": 18826,
  "body": "If I wanted to learn how to tie certain knots,
  ➔or learn about new knots and what they're used for,
  ➔what are some good resources to look up?",
  "title": "What's a good resource for learning to tie knots for climbing?",
  "post_type": "question"},
{"id": "24440",
  "parent_id": 18825,
  "body": "Knots and Ropes for Climbers by Duane Raleigh is a fantastic
  ➔illustrated resource tailored specifically to climbers. The ABoK
  ➔is great, but a but beyond the pale of what the average rock...",
  "post_type": "answer"},
{"id": "18826",
  "parent_id": 18825,
  "body": "Animated Knots By Grog Arguably the best resource online for knot
  ➔tying is Animated Knots by Grog , it's used by virtually every avid
  ➔knot tyer I've known. They have excellent step-by-step animatio...",
  "post_type": "answer"}]
```

Документ 18826 отмечен как
принятый ответ на вопрос.

Это наш доку-
мент с вопросом.

Это документ, отмеченный как принятый ответ
на вопрос (accepted_answer_id=18826).

Эти документы являются ответами на
вопрос в первом документе (id=18825).

Это документы с ответами, которые
относятся к вопросу.

В предыдущем листинге первый документ – это вопрос, наиболее связанный с запросом **climbing knots**. У вопроса есть два ответа, которые связаны с родительским вопросом через поле `parent_id` в каждом ответе. Один из этих ответов был выбран в качестве принятого ответа, что определяется установкой поля `accept_answer_id` (в данном случае 18826) в документе вопроса.

Поле `body` документов вопроса содержит пояснения к вопросу, в то время как поле `body` ответа содержит полный ответ. Только посты вопроса имеют заголовок, который является резюме вопроса. Несколько других полей (таких как `view_count`, `answer_count` и `owner_user_id`) здесь опущены, но доступны в полном наборе данных как поля метаданных, которые могут помочь с релевантностью поиска с использованием BM25 в сочетании с другими сигналами.

Теперь, когда вы ознакомились с моделью данных, давайте уделим немного времени, чтобы попробовать несколько запросов и посмотреть, какие типы вопросов возвращаются. Следующий листинг ищет вопросы, соответствующие общему запросу.

Листинг 13.3. Выполнение базового лексического поиска для climbing knots

```
def search_questions(query, verbose=False):
    request = {"query": query,
               "query_fields": ["title", "body"],
               "limit": 5,
               "return_fields": ["id", "url", "post_type", "title",
                                "body", "accepted_answer_id", "score"],
               "filters": [("post_type", "question")],
               "order_by": [("score", "desc"), ("title", "asc")]}
    response = outdoors_collection.search(**request)
    display_questions(query, response, verbose)

search_questions("climbing knots")
```

Сопоставляет запрос с полями title и body.

Ответ:

Query: climbing knots

Ranked Questions:

Question 21855: What are the four climbing knots used by Jim Bridwell?

Question 18825: What's a good resource for learning to tie knots for clim...

Question 18814: How to tie a figure eight on a bight?

Question 9183: Can rock climbers easily transition to canyoning?

Question 22477: Tradeoffs between different stopper knots

Мы видим, что это несколько релевантные заголовки для этого лексического запроса. Но это всего лишь базовый поиск по ключевым словам. Другие запросы не работают так же хорошо; например, запрос What is DEET в следующем листинге показывает очень нерелевантные результаты.

Листинг 13.4. Базовое лексическое сопоставление может дать нерелевантные результаты

```
search_questions("What is DEET?")
```

Ответ:

Query What is DEET?:

Ranked Questions:

Question 20403: What is bushcrafting?

Question 20977: What is "catskiing"?

Question 1660: What is Geocaching?

Question 17374: What is a tent skirt and what is its purpose?

Question 913: What is a buff?

Это показывает, как традиционный лексический поиск может потерпеть неудачу в обычных случаях использования естественного языка. В частности, инвертированный индекс страдает от проблемы зависимости запрос–термин. Это означает, что термины в запросе сопоставляются как строки с терминами в индексе. Вот почему вы видите сильные совпадения для того, что находится в результатах в листинге 13.4. *Смысл* запроса не понят, поэтому извлечение может быть основано только на сопоставлении строк.

Остальная часть этой главы предоставит основы, необходимые для использования трансформеров в поиске на естественном языке, а в главе 14 мы решим проблему вопрос–ответ, очевидную в листинге 13.4.

13.4.2. Тонкая настройка и семантический анализ сходства текста

Использование предварительно обученной готовой модели трансформера обычно не дает оптимальных результатов для подсказок, связанных с конкретной задачей. Это связано с тем, что первоначальное обучение проводилось в общем языковом контексте без учета конкретного варианта использования или домена. По сути, оно «не настроено», и использование моделей таким образом похоже на индексацию контента в поисковой системе без настройки на релевантность.

Чтобы реализовать весь потенциал трансформеров, их необходимо доработать для выполнения конкретной задачи. Это известно как *тонкая настройка* – процесс взятия предварительно обученной модели и ее обучения на более подходящих для цели данных для достижения конкретной цели варианта использования. Как для автозаполнения, так и для семантического поиска мы заинтересованы в тонкой настройке для выполнения задач обнаружения сходства текста.

Это приводит нас к обучающему и тестовому набору Semantic Text Similarity Benchmark (STS-B) (<https://ixa2.si.ehu.es/stswiki/>). Этот бенчмарк¹ включает отрывки, которые семантически похожи и непохожи, и они маркированы соответствующим образом. Используя этот набор данных, модель может быть настроена для повышения точности поиска ближайшего соседа между набором терминов и многими отрывками в корпусе, что будет нашим вариантом использования в этой главе.

В главе 14 мы настроим нашу собственную модель вопрос–ответ, чтобы вы могли увидеть, как это делается. Однако для наших целей в этой главе мы будем использовать проект, который уже включает предварительно настроенную модель для этой задачи: SBERT.

¹ Бенчмаркинг – это метод анализа, который позволяет рассмотреть разные решения, протестировать их производительность и сравнить полученные показатели скорости. Разработчику нужны эти полезные данные, особенно при необходимости ускорить и оптимизировать работу приложения. – *Прим. ред.*

13.4.3. Знакомство с библиотекой трансформеров SBERT

SBERT, или Sentence-BERT, – это метод и библиотека Python, основанные на трансформерах, которые построены на идее, что модель BERT может быть точно настроена таким образом, что два семантически похожих предложения, а не просто токены в векторном пространстве должны быть представлены ближе. В частности, SBERT *пулирует* (собирает в один пул данных) все эмбединги BERT в одном предложении в один вектор. (Пулирование – это причудливый способ наименования объединения значений.) После того как SBERT пулирует значения, он обучается на предмет сходства между предложениями, используя специальную нейронную сеть, которая учится оптимизироваться для задачи STS-B. Для получения дополнительных подробностей о реализации ознакомьтесь с документом «Sentence-BERT» Нильса Реймерса и Ирины Гуревич (<https://arxiv.org/abs/1908.10084>). В следующих листингах вы найдете обзор того, как использовать SBERT через библиотеку Python `sentence_transformers`. Мы начнем с импорта библиотеки с использованием предварительно обученной модели `roberta-base-nli-stsb-mean-tokens`, которая основана на архитектуре RoBERTa. Полезно думать о RoBERTa как об улучшенной и доработанной версии BERT с оптимизированными гиперпараметрами (параметрами конфигурации) и небольшими изменениями в исходных методах.

Гиперпараметры

В машинном обучении гиперпараметрами являются любые значения параметров, которые можно изменить до обучения и которые изменяют процесс обучения и повлияют на полученную модель.

К сожалению, вы часто не знаете, какие значения гиперпараметров следует установить в начале, поэтому вам, возможно, придется изучать оптимизированные значения с течением времени с помощью итераций и измерений.

В названии модели `roberta-base-nli-stsb-mean-tokens` мы также можем увидеть некоторые термины, которые вы можете не узнать, включая «nli» и «mean-tokens». NLI означает вывод естественного языка (поддомен NLP, используемый для прогнозирования языка), а `mean-tokens` относится к токенизации всего предложения, которая объединяется вместе как среднее числовых значений эмбедингов токенов. Использование смысловых токенов возвращает единый 768-мерный эмбединг для всего предложения.

Следующий листинг импортирует библиотеку `sentence_transformers`, загружает модель и отображает полную архитектуру сети.

Листинг 13.5. Загрузка модели RoBERTa SentenceTransformer

```
from sentence_transformers import SentenceTransformer
transformer = SentenceTransformer("roberta-base-nli-stsb-mean-tokens")
```

Теперь объект `transformer` PyTorch содержит архитектуру нейронной сети для трансформеров, а также все веса модели.

Загрузив нашу модель, мы можем извлечь эмбединги из текста. Вот где начинается самое интересное. Мы можем взять предложения и передать их через архитектуру нейронной сети, используя предварительно обученную модель, и получить в результате эмбединги. У нас есть четыре предложения, которые мы закодируем и оценим в следующих листингах.

Листинг 13.6 демонстрирует, как кодировать несколько фраз в плотные векторные эмбединги.

Листинг 13.6. Кодирование фраз в виде плотных векторных эмбедингов

**Четыре предложения, которые мы хотим закодировать.
Мы передадим все это для кодирования как один пакет.**

```
phrases = ["it's raining hard", "it is wet outside",  
           "cars drive fast", "motorcycles are loud"]  
embeddings = transformer.encode(phrases, convert_to_tensor=True)  
print("Number of embeddings:", len(embeddings))  
print("Dimensions per embedding:", len(embeddings[0]))  
print("The embedding feature values of 'it's raining hard':")  
print(embeddings[0])
```

Ответ:

**Просто вызывает `transformer.encode`,
и абстракция `sentence_transformers`
делает всю тяжелую работу за вас.**

Number of embeddings: 4

Dimensions per embedding: 768

The embedding feature values of "it's raining hard":

```
tensor( 1.1609e-01, -1.8422e-01, 4.1023e-01, 2.8474e-01, 5.8746e-01,  
        7.4418e-02, -5.6910e-01, -1.5300e+00, -1.4629e-01, 7.9517e-01,  
        5.0953e-01, 3.5076e-01, -6.7288e-01, -2.9603e-01, -2.3220e-01,  
        ...  
        5.1413e-01, 3.0842e-01, -1.1862e-01, 5.9565e-02, -5.5944e-01,  
        9.9763e-01, -2.2970e-01, -1.3132e+00])
```

В предыдущем листинге мы берем каждое предложение и передаем его кодировщику. Это приводит к тензору для каждого предложения. *Тензор* – это обобщаемая структура данных для хранения потенциально многомерных значений. Скаляр (отдельное значение), вектор (массив скаляров), матрица (массив векторов) или даже многомерная матрица (массив матриц, матрица из матриц и т. д.) – все это примеры тензоров различных размерностей. Тензоры создаются трансформер-кодировщиками, такими как SBERT, при кодировании текста. Для нашего варианта использования тензор в листинге 13.6 – это эмбединг, содержащий 768 измерений, представленных в виде чисел с плавающей точкой.

С нашими эмбедингами мы теперь можем считать косинусные сходства (скалярное произведение векторов, нормированных на единицу), чтобы увидеть, какие фразы являются ближайшими соседями друг к другу. Мы сравним каждую фразу с каждой другой фразой и отсортируем их по сходству, чтобы увидеть, какие из них наиболее похожи. Этот процесс

пошагово рассматривается в листингах 13.7 и 13.8. Мы будем использовать встроенную библиотеку PyTorch для вычисления скалярного произведения, чтобы выполнить эти сравнения, что позволяет нам передавать эмбединги с помощью одного вызова функции. Затем мы можем отсортировать полученные сходства и посмотреть, какие две фразы наиболее похожи друг на друга, а какие две наименее похожи. Следующий листинг вычисляет сходства между каждым из эмбедингов фраз.

Листинг 13.7. Сравнение всех фраз друг с другом

Нормирует эмбединги на единицу для скорости вычислений, поэтому скалярные произведения = косинусные сходства.

```
def normalize_embedding(embedding):
    normalized = numpy.divide(embedding, numpy.linalg.norm(embedding))
    return list(map(float, normalized))

normalized_embeddings = list(map(normalize_embedding, embeddings))
similarities = sentence_transformers.util.dot_score(normalized_embeddings,
                                                    normalized_embeddings)
print("The shape of the resulting similarities:", similarities.shape)
```

Вывод:

The shape of the resulting similarities: torch.Size([4, 4])

Мы печатаем форму объекта сходства в листинге 13.7, чтобы увидеть, сколько сравнений у нас есть. Мы имеем форму 4×4 ([4, 4]), потому что у нас есть 4 фразы, и каждая фраза имеет оценку сходства с каждой другой фразой и с собой. Все оценки сходства находятся в диапазоне от 0.0 (наименее похоже) до 1.0 (наиболее похоже). Форма включена сюда, чтобы помочь показать сложность сравнения многих фраз. Если бы было 100 фраз, форма сходства была бы 100×100. Если бы было 10 000 фраз, форма сходства была бы 10 000×10 000. Таким образом, по мере добавления фраз для сравнения вычислительные и пространственные затраты будут увеличиваться как функция n^2 , где n – количество фраз.

Вычислив сходства для наших четырех фраз, мы сортируем и печатаем их в следующем листинге.

Листинг 13.8. Сортировка по сходству и вывод результатов

```
def rank_similarities(phrases, similarities, name=None):
    a_phrases = []
    b_phrases = []
    scores = []
    for a in range(len(similarities) - 1):
        for b in range(a + 1, len(similarities)):
            a_phrases.append(phrases[a])
            b_phrases.append(phrases[b])
            scores.append(float(similarities[a][b]))
```

Получаем оценку для каждой пары.

Добавляет все пары фраз в датафрейм.

Мы не дублируем фразы и не добавляем сходство фразы к себе, так как оно всегда будет равно 1.0.

```

dataframe = pandas.DataFrame({"score": scores, "phrase a": a_phrases,
                              "phrase b": b_phrases})
dataframe["idx"] = dataframe.index
dataframe = dataframe.reindex(columns=["idx", "score",
                                      "phrase a", "phrase b"])

```

Добавляет столбец индекса.

```

return dataframe.sort_values(by=["score"],
                             ascending=False,
                             ignore_index=True)

```

Сортирует оценки (по возрастанию = False для самых высоких оценок).

```

dataframe = rank_similarities(phrases, similarities)
display(HTML(dataframe.to_html(index=False)))

```

Ответ:

idx	score	phrase a	phrase b
0	0.669060	it's raining hard	it is wet outside
5	0.590783	cars drive fast	motorcycles are loud
1	0.281166	it's raining hard	cars drive fast
2	0.280800	it's raining hard	motorcycles are loud
4	0.204867	it is wet outside	motorcycles are loud
3	0.138172	it is wet outside	cars drive fast

Теперь мы видим, что две фразы, которые наиболее похожи друг на друга, – это «идет сильный дождь» и «на улице мокро». Мы также видим сильное сходство между автомобилями и мотоциклами.

Две самые непохожие фразы – это «на улице мокро» и «машины едут быстро». Из этих примеров совершенно ясно, что этот процесс семантического кодирования работает – мы можем связать дождь с тем, что на улице мокро. Плотные векторные представления захватили контекст, и, хотя слова разные, есть общий смысл. Обратите внимание на оценки: два верхних похожих сравнения имеют оценку более 0.59, а следующее ближайшее сравнение имеет оценку менее 0.29. Это связано с тем, что только два верхних сравнения кажутся похожими друг на друга, как мы бы их воспринимали в задаче *понимания естественного языка* (NLU). Как разумные люди, мы можем сгруппировать дождь и сырость («погода»), а также можем сгруппировать автомобили и мотоциклы («транспортное средство»). Также интересно, что автомобили, скорее всего, едут медленнее, когда на земле мокро, так что это, вероятно, объясняет низкое сходство последней пары.

13.5. Автозаполнение естественного языка

Теперь, когда мы знаем, что наш процесс векторного кодирования и подобия работает хорошо, пришло время применить эту технику эмбединга в реальном поисковом сценарии – автозаполнении естественного языка!

В этом разделе мы покажем практическое использование преобразователей предложений во время поиска с базовой и быстрой реализацией семантического автозаполнения. Мы применим то, чему научи-

лись до сих пор, для извлечения понятий из набора данных Outdoor. Используя spaCy (библиотеку Python NLP, которую мы использовали в главе 5), мы разобьем существительные и глаголы на части, чтобы получить понятия (концепты) outdoor. Мы поместим эти концепты в словарь и обработаем их, чтобы получить их эмбединги. Затем будем использовать словарь в приблизительном индексе ближайшего соседа (ANN) для запроса в реальном времени. Это даст нам возможность ввести термин и получить наиболее похожие понятия, которые существуют в словаре. Наконец, мы возьмем эти понятия и представим их пользователю в порядке сходства, продемонстрировав умное, естественное автозаполнение языка. Опыт и тестирование показывают, что это работает намного лучше, чем даже хорошо настроенный подсказчик в большинстве лексических поисковых систем. Мы увидим, что он гораздо менее шумный, а также что похожие термины, которые пишутся по-разному, будут автоматически включены в предложения. Это связано с тем, что мы не сравниваем строки ключевых слов друг с другом; вместо этого мы сравниваем эмбединги, которые представляют значение и контекст. Это воплощение поиска «вещей, а не строк», как представлено в разделе 1.2.4.

13.5.1. Получение фраз существительных и глаголов для нашего словаря ближайшего соседа

Используя то, что мы узнали в главе 5, мы напишем простую функцию для извлечения *концептов* из корпуса. Мы не будем включать никакую таксономическую иерархию, и мы не будем строить здесь полный граф знаний. Нам просто нужен надежный список часто используемых существительных и глаголов.

Концепты в нашем примере – это важные «вещи» и «действия», которые люди обычно ищут. Нам также нужно понять набор данных, что лучше всего достигается путем изучения концептов и того, как они соотносятся друг с другом. Понимание корпуса имеет решающее значение при создании любого поискового приложения, и нет никаких исключений при использовании передовых методов обработки естественного языка.

В следующем листинге показана стратегия, которая обеспечит достойную качественную базу кандидатов в концепты для нашего словаря, одновременно удаляя значительный шум из результатов автозаполнения.

Листинг 13.9. Использование spaCy Matcher для получения желаемых частей текста

Исходные текстовые метки, которые были нормированы в концепты.

Загружает английскую модель spaCy NLP.

Все нормированные существительные/глагольные фразы («концепты») в корпусе.

```
nlp = spacy.load("en_core_web_sm")
phrases = []
sources = []
matcher = Matcher(nlp.vocab)
```

Использует spaCy Matcher для разбиения шаблонов на концепт-метки.

Теги частей речи, соответствующие существительным.

```
noutags = ["NN", "NNP", "NNS", "NOUN"]
verbtags = ["VB", "VBD", "VBG", "VBN", "VBP", "VBZ", "VERB"]
```

Теги частей речи, соответствующие глаголам.

Добавляет шаблон соответствия фраз существительных в конвейер анализа spaCy.

```
matcher.add("noun_phrases", [{"TAG": {"IN": noutags},
                               "IS_ALPHA": True,
                               "OP": "+"}])
matcher.add("verb_phrases", [{"TAG": {"IN": verbtags},
                              "IS_ALPHA": True, "OP": "+",
                              "LEMMA": {"NOT_IN": ["be"]}])
```

Добавляет шаблон соответствия фраз глаголов. Вы можете добавить больше шаблонов NOT_IN, чтобы исключить другие стоп-слова – глаголы.

```
for doc, _ in tqdm.tqdm(nlp.pipe(yield_tuple(dataframe,
                                             source_field="body",
                                             total=total),
                              batch_size=40,
                              n_threads=4,
                              as_tuples=True),
                        total=total):
```

Обрабатывает поле тела для каждого вопроса на открытом воздухе партиями по 40 документов с использованием 4 потоков.

```
matches = matcher(doc)
for _, start, end in matches:
    span = doc[start:end]
    phrases.append(normalize(span))
    sources.append(span.text)
```

Получает все совпадения фраз существительных и глаголов и сохраняет их в списках источников и фраз.

```
concepts = {}
labels = {}
for i, phrase in phrases:
    if phrase not in concepts:
        concepts[phrase] = 0
        labels[phrase] = sources[i]
    concepts[phrase] += 1
```

Объединяет нормированные концепты по частоте терминов.

В предыдущем листинге мы используем spaCy Matcher для обнаружения шаблонов как тегов частей речи. Мы также явно удаляем формы глагола «to be» из концептов глаголов. Глагол «to be» часто используется во многих бесполезных ситуациях и часто загромождает концепт-предложения. Мы могли бы еще больше улучшить качество, удалив другие шумные глаголы, такие как «have» и «can», но это всего лишь пример на данный момент. В этом листинге также представлен языковой конвейер SpaCy (`nlp.pipe`). Функция `pipe` принимает размер пакета и количество потоков для использования в качестве параметров, а затем выполняет потоковую обработку текста параллельными пакетами (это быстрее, чем делать отдельные вызовы для каждого документа).

С помощью функции в листинге 13.9 мы теперь можем получить список концептов. При запуске на вашем компьютере это может занять некоторое время, поэтому будьте терпеливы. Следующий листинг возвращает наиболее важные концепты и метки из коллекции `outdoor`.

Листинг 13.10. Генерация наиболее частых концептов в корпусе

```

collection = engine.get_collection("outdoors")
concepts, labels = get_concepts(collection, source_field="body",
                                load_from_cache=True)
topcons = {key: value for (key, value)
            in concepts.items() if value > 5}
print("Total number of labels:", len(labels.keys()))
print("Total number of concepts:", len(concepts.keys()))
print("Concepts with greater than 5 term frequency:", len(topcons.
keys()))
print(json.dumps(topcons, indent=2))

```

Ответ:
 Total number of labels: 124366
 Total number of concepts: 124366
 Concepts with greater than 5 term frequency: 12375
 {
 "have": 32782,
 "do": 26869,
 "use": 16793,
 ...
 "streamside vegetation": 6,
 "vehicle fluid": 6,
 "birdshot": 6
 }

Помимо получения концептов для набора данных outdoors, листинг 13.10 отфильтровал весь набор данных до topcons, который включает только концепты с частотой больше 5. Фильтрация ограничит шум от терминов, которые не так часто встречаются в корпусе, таких как орфографические ошибки и редкие термины, которые мы не хотим предлагать в сценарии автозаполнения.

13.5.2. Получение эмбедингов

Мы собираемся выполнить сложное нормирование, которое нормирует схожие связанные концепты. Но вместо алгоритмического нормирования (например, стемминга) мы нормируем плотное векторное пространство из 768 измерений признаков. Подобно стеммингу, цель этого – повысить *отзыв* (процент успешно возвращенных релевантных документов, recall). Но вместо использования стеммера мы будем находить и сопоставлять вместе тесно связанные концепты. Напоминаем, что мы нормируем только существительные и глагольные фразы. Игнорирование других слов похоже на удаление стоп-слов, но это нормально, потому что мы хотим предложить похожие концепты как можно более кратко. У нас также будет гораздо лучшее представление контекста и значения оставшихся фраз. Поэтому окружающие несуществительные и неглагольные термины подразумеваются.

Теперь, когда у нас есть список концептов (из последнего раздела), мы обработаем их с помощью нашей загруженной model (модель RoBERTa Sentence Transformer, которую мы загрузили в листинге 13.5) для из-

влечения эмбедингов. Это может занять некоторое время, если у вас нет графического процессора, поэтому после того, как мы вычислим эмбединги в первый раз, мы сохраним их в «pickle file» («маринад», сериализованный объект Python, который можно легко сохранять и записывать на диск и загружать с него, позволяет компактно и эффективно хранить сложные структуры данных, такие как списки, словари и экземпляры классов). Если вы когда-нибудь захотите перезапустить блокнот, можете просто загрузить ранее созданный файл pickle и не тратить еще полчаса на повторную обработку необработанного текста.

Предупреждение о гиперпараметрах! Термин `minimum_frequency` является гиперпараметром, и в следующем листинге он установлен на уровень больше пяти (≥ 6), чтобы минимизировать шум от редких терминов. Мы встретим больше гиперпараметров в других листингах в этой и следующей главе, особенно когда займемся тонкой настройкой. После того как вы ознакомитесь с остальными листингами в этой главе, мы рекомендуем вам вернуться и изменить значение `minimum_frequency` и посмотреть, как оно изменяет полученные результаты. Вы можете найти значение, которое будет более подходящим и точным, чем то, к которому мы пришли здесь.

Листинг 13.11. Извлечение эмбедингов нашего словаря концептов

```
def get_embeddings(texts, model, cache_name, ignore_cache=False):
    ...
    embeddings = model.encode(texts)
    ...
    return embeddings
```

Кеширующий код удален для краткости.

Это гиперпараметр! Мы игнорируем термины, которые встречаются меньше этого количества раз во всем корпусе. Снижение этого порога может снизить точность, а повышение – отзыв.

```
minimum_frequency = 6
phrases = [key for (key, tf) in concepts.items() if tf >= minimum_frequency]
cache_name = "outdoors_embeddings"
embeddings = get_embeddings(phrases, transformer,
                           cache_name, ignore_cache=False)

print(f"Number of embeddings: {len(embeddings)}")
print(f"Dimensions per embedding: {len(embeddings[0])}")
```

Ответ:

```
Number of embeddings: 12375
Dimensions per embedding: 768
```

Из листинга 13.11 вы можете видеть, что один эмбединг был сгенерирован из каждого из наших 12 375 концептов. Все эмбединги имеют одинаковую размерность из одного и того же плотного векторного пространства и, следовательно, могут напрямую сравниваться друг с другом.

Рисунок 13.6 демонстрирует, как выглядят эти эмбединги и как они соотносятся друг с другом при построении в 3D.

Визуализация всех эмбедингов фраз-концептов

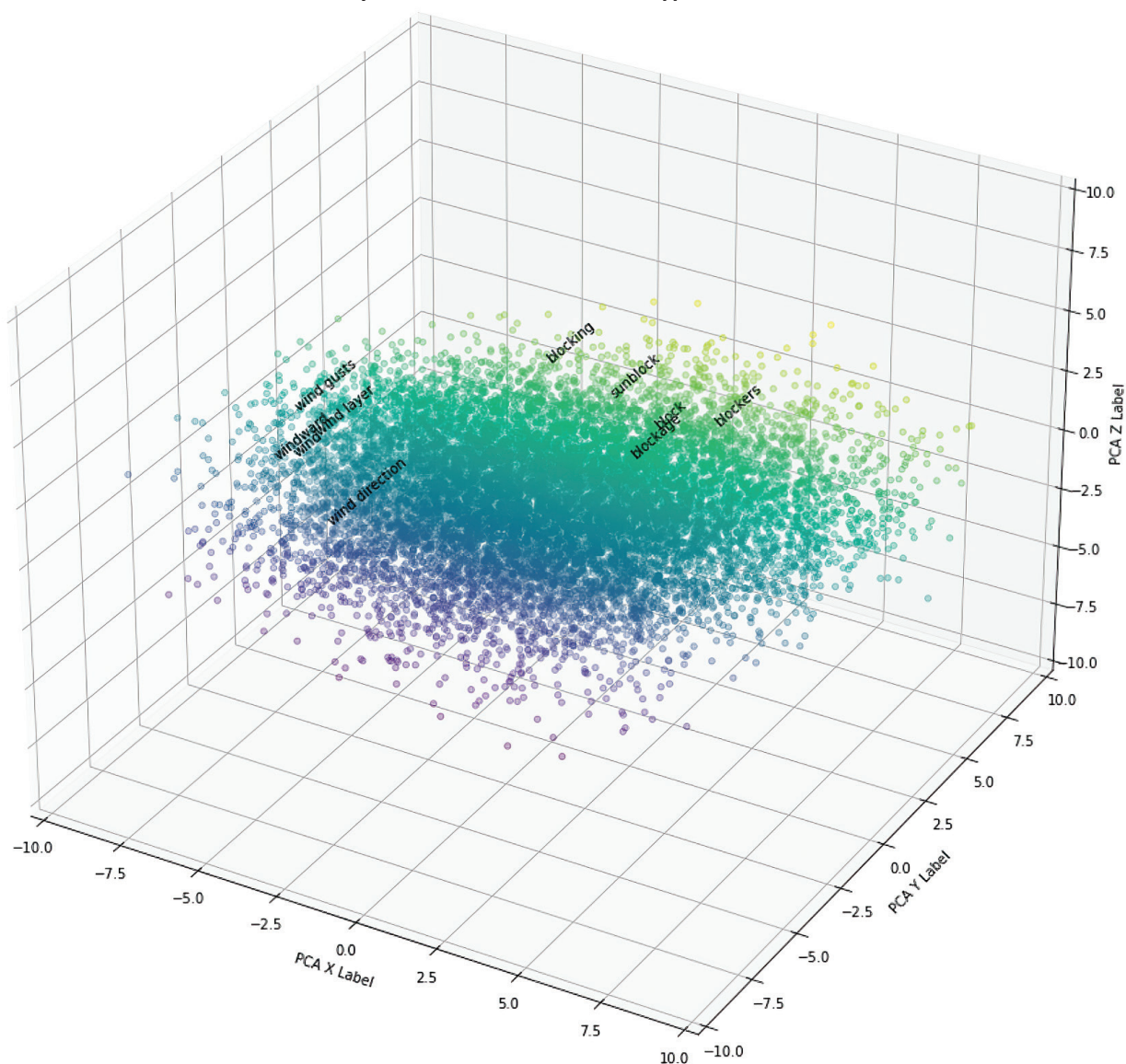


Рис. 13.6. Векторные пространства для эмбедингов концептов, преобразованные в 3D-визуализации

Сходства некоторых концептов на рисунке были маркированы, чтобы показать соседство смысла. Концепты, связанные с «ветром» и «блоком», иллюстрируют, где они расположены относительно друг друга в векторном пространстве. Мы использовали уменьшение размерности, чтобы уменьшить 768 измерений для каждого эмбединга до 3 измерений (x , y , z), чтобы их можно было легко нарисовать. *Уменьшение размерности* – это метод уплотнения одного вектора с большим количеством признаков в другой вектор с меньшим количеством признаков. Во время этого уменьшения отношения в векторном пространстве сохраняются в максимально возможной степени.

При уменьшении размерности теряется контекст

При выполнении уменьшения размерности теряется много контекста, поэтому визуализация на рис. 13.6 представлена только для того, чтобы дать вам интуитивное представление о векторном пространстве и сходстве понятий, а не для того, чтобы предположить, что уменьшение до трех измерений – идеальный способ представления понятий.

С эмбедингами, рассчитанными из листинга 13.11, мы теперь можем выполнить масштабное сравнение, чтобы увидеть, какие термины более тесно связаны друг с другом. Мы сделаем это, вычислив косинусное сходство – скалярное произведение для каждого нормированного на единицу эмбединга, связанного с каждым другим единичным эмбедингом. Обратите внимание, что мы ограничиваем количество эмбедингов, сравниваемых в этом примере, потому что количество вычислений, которые необходимо выполнить, растет экспоненциально по мере увеличения количества эмбедингов. Если вы не уверены, что мы имеем в виду, давайте сделаем несколько быстрых математических расчетов. Каждый эмбединг имеет размерность 768 значений с плавающей точкой. Сравнение 250 лучших эмбедингов друг с другом дает $250 \times 250 \times 768 = 48\,000\,000$ вычислений с плавающей точкой. Если бы мы сравнили полный список из 12 375 эмбедингов, то это составило бы $12\,375 \times 12\,375 \times 768 = 17\,612\,000\,000$ вычислений с плавающей точкой. Это не только обрабатывалось бы чудовищно долго, но и потребовало бы немыслимо много памяти.

В следующем листинге мы выполним сравнение методом грубой силы¹ 250 лучших понятий для оценки распределения оценок сходства.

Листинг 13.12. Изучение оценок сходства из заголовка словаря

```
normalized_embeddings = list(map(normalize_embedding, embeddings))
similarities = sentence_transformers.util.dot_score(
    normalized_embeddings[0:250],
    normalized_embeddings[0:250])
comparisons = rank_similarities(phrases, similarities)
display(HTML(comparisons[:10].to_html(index=False)))
```

Находит пары с наивысшими оценками скалярного произведения.

Ранжирует сходства, как определено в листинге 13.8.

Ответ:

idx	score	phrase a	phrase b
31096	0.928151	protect	protection

¹ Метод грубой силы в программировании представляет собой прямой подход к решению задачи путем перебора всех возможных вариантов. Он используется для многих элементарных, но важных алгоритмических задач, таких как вычисление суммы чисел, поиск наибольшего элемента в списке, пузырьковая сортировка. – *Прим. ред.*

13241	0.923570	climbing	climber
18096	0.878894	camp	camping
...			
7354	0.782962	climb	climber
1027	0.770643	go	leave
4422	0.768611	keep	stay

Как вы можете видеть в листинге 13.12, датафрейм `scores` теперь содержит отсортированный список всех фраз, сравниваемых друг с другом, причем наиболее похожими являются «protect» и «protection» со скалярным произведением сходства 0.928.

Обратите внимание, что индекс 250 является произвольным и может быть изменен на большие значения для визуализации большего количества данных. Помните, что мы узнали в листинге 13.7: использование `n` концептов дает тензор `[n, n]` в форме. Это дает в общей сложности $250 \times 250 = 62500$ сходств для примера в листинге 13.12.

Следующий листинг отображает распределение 250 лучших оценок сравнения сходства концептов.

Листинг 13.13. Распределение сходства слов

```
from plotnine import *
candidate_synonyms = comparisons[comparisons["score"] > 0.0]
{
    ggplot(comparisons, aes("idx", "score")) +
        geom_violin(color="blue") +
        scale_y_continuous(limits=[-0.4, 1.0],
                           breaks=[-0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8, 1.0])
}
```

Вывод листинга 13.13 показан на рис. 13.7, и распределение проясняет процент концептов, которые наиболее связаны друг с другом. Вы видите, что очень немногие сравнения имеют оценку сходства больше 0.6, а подавляющее большинство имеет оценку сходства меньше этого.

Мы построили график распределения оценок, чтобы иметь возможность оценить их и использовать нашу интуицию для выбора базового порога сходства во время запроса (используется далее в листинге 13.15). Визуализация на рис. 13.7 очень многообещающая. Поскольку большинство концептов отмечены как несхожие, мы можем надежно выбрать достаточно высокое число в качестве порога качественных предложений (например, 0.6 в этом примере). Когда мы выполняем автозаполнение во время поиска, нам интересно увидеть только первые пять-десять предложенных терминов, поэтому это распределение показывает, что мы можем сделать это надежно.

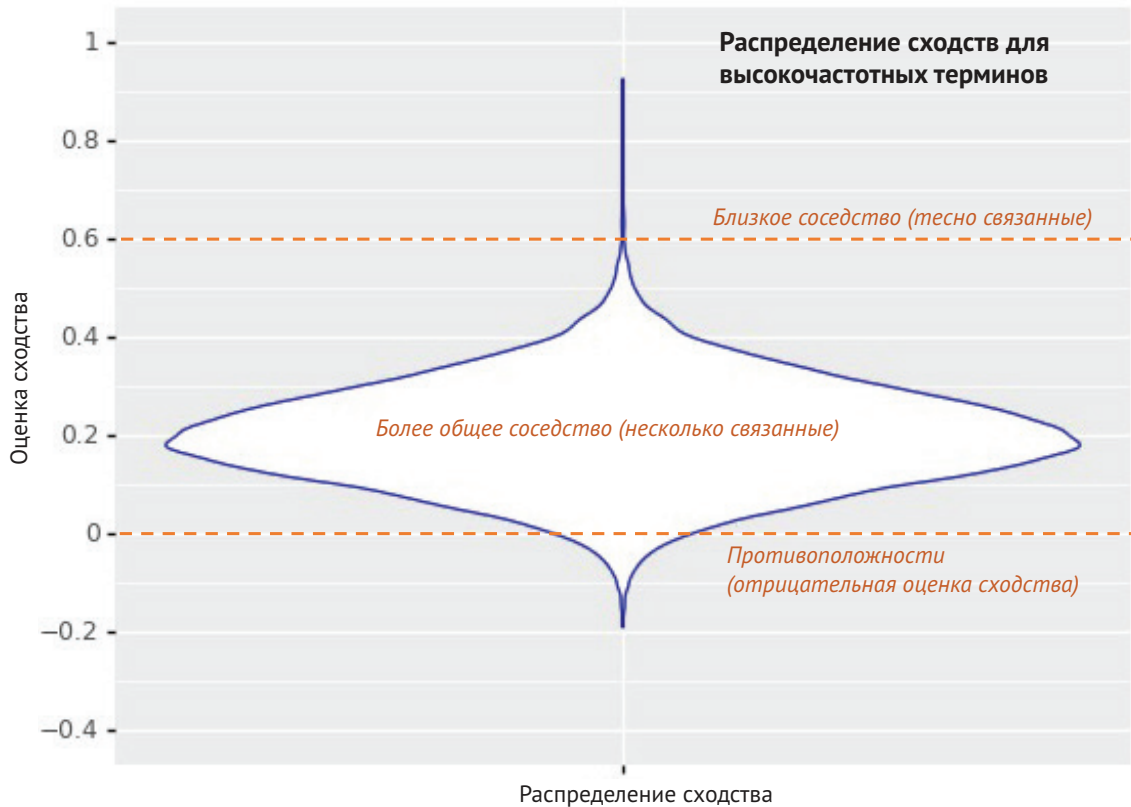


Рис. 13.7. Распределение того, как топ-250 концептов оцениваются по показателю сходства скалярного произведения при сравнении друг с другом. Обратите внимание, что очень немногие сравнения дают оценку выше 0.6, а большинство оценок меньше 0.4 (очень низкая достоверность)

13.5.3. Приближенный поиск ближайших соседей

Перед реализацией работающего автозаполнения нам нужно решить еще одну важную проблему. Проблема в том, что во время запроса мы в идеале не хотим сравнивать каждый поисковый термин со всеми 12 375 другими терминами. Это было бы неэффективно и медленно из-за размерности и вычислительных затрат на использование функции `sentence_transformers.util.dot_score`. Даже если бы мы хотели вычислить сходства скалярного произведения для всех наших документов, это бы становилось все медленнее и медленнее по мере масштабирования до миллионов документов, поэтому в идеале мы бы оценивали только документы, которые имеют высокую вероятность быть похожими.

Мы можем достичь этой цели, выполнив так называемый *приближенный поиск ближайшего соседа* (ANN). Поиск ANN эффективно вернет наиболее тесно связанные документы при задании вектора, без накладных расходов на вычисление сходств эмбедингов по всему корпусу. Поиск ANN призван пожертвовать некоторой точностью в обмен на улучшенную логарифмическую вычислительную затратность, а также эффективность использования памяти и дискового пространства.

Для реализации нашего поиска ANN мы будем использовать стратегию оптимизации времени индексации для хранения векторов контента, доступных для поиска, заранее в специализированной структуре данных. Думайте о поиске ANN как о «перевернутом индексе» для плотного векторного поиска.

Для наших целей мы будем использовать *графы HNSW* (Hierarchical Navigable Small World) для индексации и запроса наших плотных векторов. Мы также рассмотрим другие подходы, основанные на кластеризации и хешировании, такие как квантизация продукта и инвертированные файловые индексы (IVF), в разделе 13.7. HNSW описан в аннотации исследовательской работы «Эффективный и надежный приближенный поиск ближайшего соседа с использованием графов Hierarchical Navigable Small World», написанной Ю. А. Малковым и Д. А. Яшуниным (<https://arxiv.org/abs/1603.09320>):

Мы представляем новый подход к приближенному поиску k-ближайших соседей на основе навигационных графов малых миров с контролируемой иерархией (иерархический NSW, HNSW).... Иерархический NSW постепенно строит многослойную структуру, состоящую из иерархических наборов графов близости (слоев) для вложенных подмножеств сохраненных предметов.

Это означает, что HNSW будет собирать похожие векторы в кластеры по мере построения индекса. Навигационные графы HNSW работают, организуя данные в соседства (районы) и соединяя эти соседства вероятными ребрами взаимоотношений. Когда индексируется плотное векторное представление, наиболее подходящее соседство и его потенциальные связи определяются и сохраняются в структуре данных графа.

Различные подходы ANN

В этой главе мы используем алгоритм HNSW для поиска ANN. HNSW обеспечивает отличный баланс между отзывом и пропускной способностью запросов и в настоящее время (на момент написания книги) является одним из самых популярных методов ANN. Однако существует множество других подходов ANN, включая гораздо более простые методы, такие как локально-чувствительное хеширование (LSH). LSH разбивает векторное пространство на хеш-контейнеры (представляющие соседства в векторном пространстве) и кодирует (хеширует) каждый плотный вектор в один из этих контейнеров. Хотя отзыв, как правило, намного выше для HNSW по сравнению с LSH, HNSW зависит от ваших данных для генерации соседств, и соседства могут со временем меняться, чтобы лучше соответствовать вашим данным. Соседства LSH (хеши) генерируются независимым от данных способом, что может лучше соответствовать некоторым вариантам использования, требующим априорного сегментирования в распределенных системах. Мы также рассмотрим другие подходы, основанные на кластеризации и хешировании, такие как квантизация продукта и инвертированные файловые индексы (IVF), в разделе 13.7. Возможно, вам стоит изучить различные алгоритмы ANN, чтобы найти тот, который лучше всего подходит для вашего приложения.

Когда поиск HNSW инициируется с использованием плотного векторного запроса, он находит лучшую точку входа в кластер для запроса и ищет ближайших соседей. Существует много других методов оптимизации, которые реализует HNSW, и мы рекомендуем вам прочитать статью, если вы хотите узнать больше.

13.5.4. Реализация индекса ANN

Для реализации нашего поиска ANN мы начнем с использования библиотеки под названием Non-Metric Space Library (NMSLIB). Эта библиотека включает каноническую реализацию алгоритма HNSW.

Мы выбрали эту библиотеку не только потому, что она быстрая, но и потому, что она проста в использовании и требует очень мало кода. Apache Lucene также включает в себя тип плотного векторного поля с собственной поддержкой HNSW, что делает алгоритм доступным в Solr, OpenSearch, Elasticsearch и других движках на основе Lucene, таких как MongoDB Atlas Search. Реализация HNSW также доступна в других поисковых системах, таких как Vespa.ai, Weaviate, Milvus и др.

NMSLIB надежна, хорошо протестирована и широко используется для приложений ANN. NMSLIB также подходит для демонстрации простоты поиска ANN, если не вдаваться в подробности реализации. Существует множество других библиотек ANN, и мы рекомендуем вам изучить некоторые из них, перечисленные на превосходном сайте ANN Benchmarks: <https://ann-benchmarks.com>.

Чтобы начать использовать NMSLIB, нам нужно просто импортировать библиотеку, инициализировать индекс, добавить все наши эмбединги в индекс в виде пакета, а затем зафиксировать¹. Автозаполнение – идеальный вариант использования при построении индекса таким образом, поскольку словарь редко обновляется. Несмотря на то что NMSLIB и другие библиотеки могут страдать от производительности времени записи в определенных ситуациях, это не повлияет на наше приложение автозаполнения с большим объемом чтения. С практической точки зрения мы можем обновлять наш индекс в автономном режиме как вечернюю или выходную работу и развертывать в производстве, когда это уместно.

Следующий листинг создает наш индекс HNSW из всех 12 375 эмбедингов, а затем выполняет пример поиска для понятий, похожих на термин «bag».

¹ Commit в программировании – это команда в системе контроля версий Git, которая фиксирует изменения в репозитории. В переводе с английского commit означает «зафиксировать». Когда пользователь использует ее, создается «снимок» текущего состояния проекта, включая все внесенные изменения. Каждое такое сохранение имеет уникальный идентификатор, который позволяет отслеживать историю изменений в проекте. – Прим. ред.

Листинг 13.14. ANN-поиск с использованием NMSLIB

```

import nmslib
concepts_index = nmslib.init(method="hsw",
                             space="negdotprod")
normalized_embeddings = list(map(normalize_embedding, embeddings))
concepts_index.addDataPointBatch(normalized_embeddings)
concepts_index.createIndex(print_progress=True)

ids, _ = concepts_index.knnQuery(
    normalized_embeddings[25], k=10)
matches = [labels[phrases[i]].lower() for i in ids]
display(matches)

```

Инициализирует новый индекс, используя граф HNSW в метрическом пространстве negdotprod (функция расстояния равна $-1 * \text{dot_product}$).

Все эмбединги могут быть добавлены в один пакет.

Получает к лучшим ближайших соседей для термина запроса «bag» (эмбединг 25) в наших эмбедингах.

Ищет метку для каждого термина.

Фиксирует индекс в памяти. Это необходимо сделать до того, как вы сможете запросить ближайших соседей.

Ответ:

```

['bag', 'bag ratings', 'bag cover', 'bag liner', 'garbage bags', 'wag bags',
 'bag cooking', 'airbag', 'paper bag', 'tea bags']

```

С созданным и зафиксированным индексом мы запустили небольшой пример, сравнивая термин «bag» и наблюдая, что возвращается. Интересно, что все эти термины являются гипонимами, что показывает еще один идеальный результат. Мы заинтересованы в том, чтобы предлагать пользователю более точные термины во время автозаполнения. Это имеет большую вероятность предоставить пользователю возможность выбрать термин, наиболее тесно связанный с его конкретной информационной потребностью. Подтвердив работоспособность нашего индекса, мы теперь можем построить простую функцию запроса, которая принимает любой термин и возвращает лучшие предложения. SBERT был обучен с использованием техники, которая кодирует похожие термины в похожие векторные эмбединги. Важно отметить, что, в отличие от большинства реализаций лексического автозаполнения, эта функция принимает любой запрос независимо от того, есть ли он уже в нашем словаре. Сначала мы берем запрос и извлекаем эмбединги, пропуская запрос через тот же кодировщик SBERT, который мы использовали для индексации наших документов. С помощью этих эмбедингов мы получаем доступ к ближайшим соседям из индекса. Если оценка сходства больше 0.75, мы считаем это совпадением и включаем это в качестве предложения. С помощью этой функции мы можем получать предложения для полных терминов, таких как «mountain hike» (горный поход), а также префиксов, таких как «dehyd-» (дегидр-).

В листинге 13.15 показана наша реализация функции автозаполнения `semantic_suggest`, которая выполняет поиск понятий с помо-

щью ANN. Наш запрос может отсутствовать в словаре, но можно получить эмбединги по требованию. Мы будем использовать порог $\text{dist} \geq 0.75$, чтобы возвращать только похожие термины, для которых мы видим высокую достоверность в сходстве.

Выберите хороший порог сходства

Мы пришли к порогу 0.75 для листинга 13.15, посмотрев на распределение из рис. 13.7. Его следует дополнительно настроить, посмотрев на качество результатов для ваших реальных пользовательских запросов.

ПРИМЕЧАНИЕ. Эта функция может чрезмерно перегрузить CPU, поэтому мы рекомендуем измерять время работы и выбирать оборудование соответствующим образом.

Листинг 13.15. Кодирование запроса и возврат понятий k-ближайших соседей

Мы устанавливаем $k=20$ для иллюстративных целей. В производственном приложении это, скорее всего, будет установлено где-то от 5 до 10.

```
def embedding_search(index, query, phrases, k=20,
                    min_similarity=0.75):
    matches = []
    query_embedding = transformer.encode(query)
    query_embedding = normalize_embedding(query_embedding)
    ids, distances = index.knnQuery(query_embedding, k=k)
    for i in range(len(ids)):
        similarity = distances[i] * -1
        if similarity >= min_similarity:
            matches.append((phrases [ids[i]], similarity))
        if not len(matches):
            matches.append((phrases [ids[1]], distances[1] * -1))
    return matches

def semantic_suggest(prefix, phrases):
    matches = embedding_search(concepts_index, prefix, phrases)
    print_labels(prefix, matches)

semantic_suggest("mountain hike", phrases)
semantic_suggest("dehyd", phrases)
```

Получает эмбединги для запроса.

Преобразует отрицательное скалярное произведение в положительное.

Мы возвращаем только термины с уровнем сходства 0.75 или выше.

Соседи не найдены! Возвращается только исходный термин.

Ответ:

Results for: mountain hike

1.000	mountain hike
0.975	mountain hiking
0.847	mountain trail
0.787	mountain guide

0.779	mountain terrain
0.775	mountain climbing
0.768	mountain ridge
0.754	winter hike

Results for: "dehyd"

0.941	dehydrate
0.931	dehydration
0.852	rehydration
...	
0.812	hydrate
0.788	hydration pack
0.776	hydration system

Мы сделали это! Теперь можем эффективно обслуживать семантическое автозаполнение на основе трансформерных эмбедингов и приблизительного поиска ближайшего соседа.

В целом качество результатов для многих запросов с этой моделью весьма впечатляет. Но будьте осторожны, крайне важно использовать маркированный набор данных для измерения успеха, прежде чем разворачивать такое решение для реальных клиентов. Мы продемонстрируем этот процесс использования маркированных данных для измерения и повышения релевантности при внедрении систем вопрос-ответ в главе 14.

13.6. Семантический поиск с эмбедингами LLM

Используя то, что мы узнали до сих пор, выведем поиск по плотным векторам на новый уровень: мы собираемся запросить эмбединги документа с эмбедингами запроса в качестве шага отзыва во время поиска.

Мы специально начали с автозаполнения в качестве нашей первой реализации, потому что это было полезно для понимания основ лингвистического сходства. Важно развить сильную интуицию о том, почему вещи похожи или не похожи в векторном пространстве. В противном случае вас будут бесконечно преследовать проблемы отзыва при использовании эмбедингов. Чтобы сформировать эту интуицию, мы начали с сопоставления и оценки основных понятий длиной всего в несколько слов каждое.

Поняв это, давайте перейдем к сравнению целых предложений. Мы собираемся выполнить семантический поиск заголовков. Помните, что мы ищем в наборе данных Stack Exchange outdoors, поэтому заголовки документов на самом деле являются резюме вопросов, задаваемых участниками. В качестве бонуса мы можем использовать ту же реализацию из последнего раздела для поиска заголовков вопросов, которые похожи друг на друга.

Эта функция будет в основном повторять функции кодирования и сходства из предыдущего раздела. Код в этом разделе еще короче, так как нам не нужно извлекать концепты.

Вот шаги, которые нужно выполнить:

- 1 получить эмбединги для всех заголовков в наборе данных outdoors;
- 2 создать индекс NMSLIB с эмбедингами;
- 3 получить эмбединги для запроса;
- 4 поиск в индексе NMSLIB;
- 5 показать заголовки ближайшего соседа.

13.6.1. Получение заголовков и их эмбедингов

Наш индекс NMSLIB будет состоять из эмбедингов заголовков. Мы используем ту же самую функцию из предыдущего примера автозаполнения, но вместо преобразования концептов мы теперь преобразуем заголовки всех вопросов, которые задавало сообщество Outdoor. Следующий листинг показывает процесс кодирования названий в эмбединги.

Листинг 13.16. Кодирование заголовков в эмбединги

```
outdoors_dataframe = load_dataframe("data/outdoors/posts.csv")
titles = outdoors_dataframe.rdd.map(lambda x: x.title).collect()
titles = list(filter(None, titles))
embeddings = get_embeddings(titles, cache_name)
```

Получает заголовки для каждого вопроса в корпусе Outdoor.

```
print(f"Number of embeddings: {len(embeddings)}")
print(f"Dimensions per embedding: {len(embeddings[0])}")
```

Получает эмбединги для заголовков (это занимает некоторое время при первом запуске, пока они не будут кешированы).

Ответ:

```
Number of embeddings: 5331
Dimensions per embedding: 768
```

Мы закодировали 5331 заголовок в эмбединги, и на рис. 13.8 показано распределение сходства их эмбедингов.

Сравните рис. 13.8 с распределениями сходства концептов из рис. 13.7. Обратите внимание на немного отличающуюся форму и распределение оценок из-за разницы между заголовками и концептами. На рис. 13.7 сверху более длинное «острие». Это связано с тем, что заголовки более конкретны и, следовательно, будут соотноситься иначе, чем более широкие существительные и глагольные фразы.

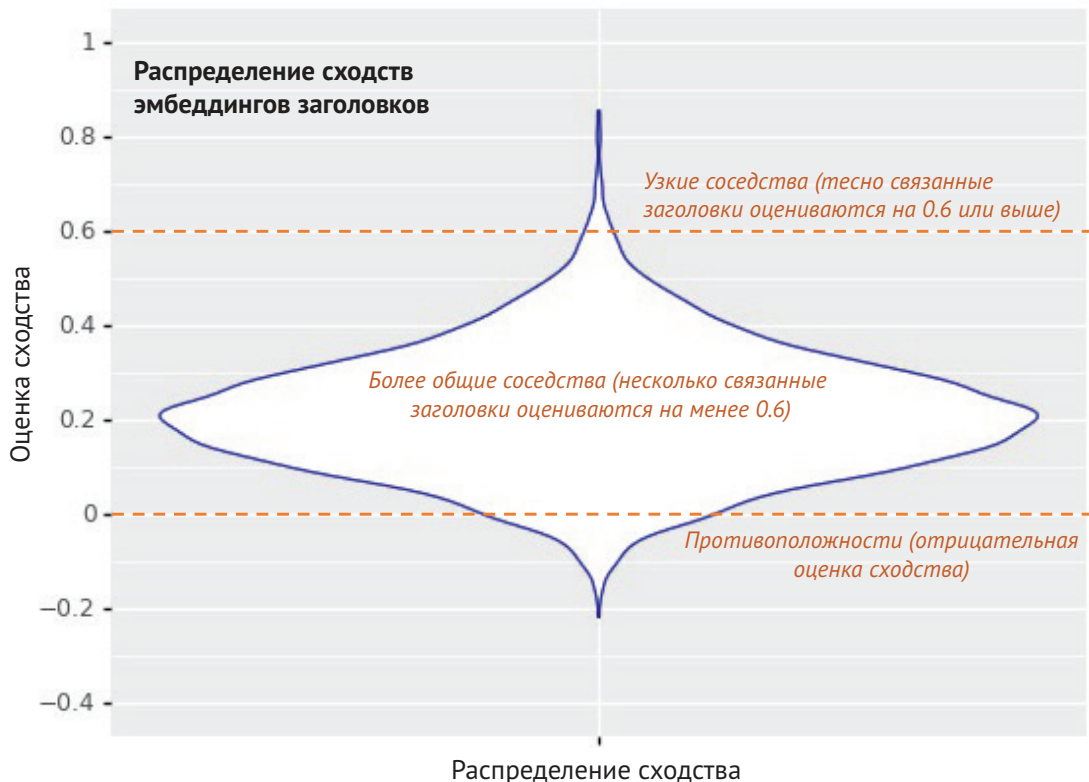


Рис. 13.8. Распределение оценок сходства при сравнении всех эмбеддингов заголовков друг с другом

13.6.2. Создание и поиск индекса ближайшего соседа

Теперь, когда мы сгенерировали эмбеддинги для всех заголовков вопросов в корпусе, можно легко создать индекс ближайшего соседа.

Листинг 13.17. Создание индекса эмбеддингов заголовков ANN

```
import nmslib
titles_index = nmslib.init(method="hns", space="negdotprod")
normalized_embeddings = list(map(normalize_embedding, embeddings))
titles_index.addDataPointBatch(normalized_embeddings)
titles_index.createIndex(print_progress=True)
```

С нашим недавно созданным индексом поиск становится простым! В листинге 13.18 показана новая функция `semantic_search`, которая реализует поиск ANN для заголовков вопросов по запросу. Это очень похоже на `semantic_suggest` из листинга 13.15, который мы реализовали для автозаполнения, — главное отличие в том, что здесь базовый индекс эмбеддинга состоит из содержимого `title`, а не из концептов, извлеченных из содержимого `body`.

Листинг 13.18. Выполнение семантического поиска заголовков

```
def semantic_search(query, phrases):
    results = embedding_search(titles_index, query, phrases,
                              k=5, min_similarity=0.6)
    print_labels(query, results)
```

embedding_search из листинга 13.15.

```
semantic_search("mountain hike", titles)
```

Ответ:

Results for: mountain hike

```
0.723 | How is elevation gain and change measured for hiking trails?
0.715 | How do I Plan a Hiking Trip to Rocky Mountain National Park, CO
0.698 | Hints for hiking the west highland way
0.694 | New Hampshire A.T. Section Hike in May? Logistics and Trail Condi...
0.678 | Long distance hiking trail markings in North America or parts the...
```

Теперь давайте на минутку задумаемся над этими результатами. Все ли они релевантны? Да – все они абсолютно связаны с запросом `mountain hike`. Но – и это очень важно – являются ли они *наиболее* релевантными документами? Мы не знаем! Причина, по которой мы не знаем, заключается в том, что `mountain hike` вообще не предоставляет много контекста. Таким образом, хотя все заголовки семантически похожи на запрос, у нас недостаточно информации, чтобы знать, являются ли они документами, которые мы должны показать пользователю.

Тем не менее очевидно, что этот подход к поиску на основе эмбединга приносит новые интересные возможности в наш набор инструментов сопоставления и ранжирования, предоставляя возможность понятийно связывать результаты. Будут ли эти результаты лучше или нет, зависит от контекста.

До сих пор мы реализовали поиск по плотным векторам в этой главе вручную, полагаясь на библиотеку NMSLIB для выполнения тяжелой работы, но в остальном показывая вам, как создать плотный векторный индекс с поддержкой ANN (HNSW) и запросить его. Мы сделали это намеренно, чтобы помочь вам понять внутреннюю работу плотного векторного поиска. Однако в вашей производственной системе вы, скорее всего, будете использовать встроенную поддержку вашей поисковой системы для плотного векторного поиска. В следующем листинге мы перейдем к использованию нашего интерфейса коллекции для реализации той же функциональности семантического поиска с использованием вашей настроенной поисковой системы или векторной базы данных.

Листинг 13.19. Выполнение векторного поиска с настроенным поисковым движком

```
def display_results(query, search_results):
    print_labels(query, [(d["title"], d["score"])
                        for d in search_results])

def index_outdoor_title_embeddings():
    create_view_from_collection(engine.get_collection("outdoors"),
                              "outdoors")
    outdoors_dataframe = spark.sql("""SELECT id, title FROM outdoors
                                   WHERE title IS NOT NULL""")
```

Создает новую коллекцию с документами, содержащими их заголовки и эмбединги.

```

ids = outdoors_dataframe.rdd.map(lambda x: x.id).collect()
titles = outdoors_dataframe.rdd.map(lambda x: x.title).collect()
embeddings = list(
    map(normalize_embedding,
        get_embeddings(titles, cache_name)))
embeddings_dataframe = spark.createDataFrame(
    zip(ids, titles, embeddings),
    schema=["id", "title", "title_embedding"])

collection = engine.create_collection("outdoors_with_embeddings")
collection.write(embeddings_dataframe)
return collection

def semantic_search_with_engine(collection, query, limit=10):
    query_vector = transformer.encode(query)
    query_vector = normalize_embedding(query_vector)
    request = {"query": query_vector,
               "query_fields": ["title_embedding"],
               "return_fields": ["title", "score", "title_embedding"],
               "quantization_size": "FLOAT32",
               "limit": limit}
    response = collection.search(**request)
    return response["docs"]

embeddings_collection = index_outdoor_title_embeddings()

query = "mountain hike"
search_results = semantic_search_with_engine(embeddings_collection,
query)
display_results(query, search_results)

```

Вычисляет нормированные эмбединги для всех документов.

Возвращает документы, выполняя поиск с помощью запроса к title_embedding.

Строковый запрос кодируется и нормируется, а затем встраивается в запрос поиска векторов.

Размер квантизации эмбедингов, который в данном случае составляет 32 бита.

Ответ:

```

0.723 | How is elevation gain and change measured for hiking trails?
0.715 | How do I Plan a Hiking Trip to Rocky Mountain National Park, CO
0.698 | Hints for hiking the west highland way
0.694 | New Hampshire A.T. Section Hike in May? Logistics and Trail Condi...
0.678 | Long distance hiking trail markings in North America or parts the...

```

Каждая поисковая система или векторная база данных имеет свои собственные уникальные API для реализации поиска по ключевым словам, поиска векторов, гибридного поиска по ключевым словам и векторам и других возможностей. Функция `semantic_search_with_engine` в листинге 13.19 демонстрирует использование независимого от движка интерфейса для запроса настроенной поисковой системы, хотя вам может показаться более мощным выполнять определенные операции с использованием API вашей системы напрямую для более сложных вариантов использования.

Ранее в этой главе мы использовали NMSLIB, чтобы помочь вам лучше понять внутреннюю работу плотного поиска векторов. Од-

нако, если вы не делаете что-то очень индивидуальное, вы, скорее всего, захотите использовать встроенную масштабируемую поддержку вашего поискового движка для плотного векторного поиска, а не реализовывать его вручную локально с помощью библиотеки вроде NMSLIB, FAISS (которую мы рассмотрим позже в этой главе) или NumPy. Вы заметите, что листинг 13.19 возвращает точно такие же результаты из вашего движка, как и реализация NMSLIB, возвращенная в листинге 13.18.

Реранкинг результатов, найденных с плотным векторным сходством

В листинге 13.18 мы выбрали пороговое значение `min_similarity` по умолчанию, т. е. оценку сходства 0.6 или выше. Изучите распределения сходства заголовков на рис. 13.8 – вы бы изменили это число, чтобы оно было отличным от 0.6?

Вы можете установить `min_similarity` на значение ниже 0.6, чтобы потенциально увеличить отзыв, и изменить `k` на значение выше 5 в качестве размера окна реранкинга (например, 250). Затем, используя этот большой набор результатов, можете выполнить реранкинг с использованием сходства скалярного произведения. Используя то, что вы узнали в главах 10–12, вы также можете включить плотное векторное сходство в качестве признака (потенциально среди многих) в более сложную модель обучения ранжированию.

Оценка сходства эмбедингов является одним из многих признаков в проработанном поисковом стеке на основе ИИ. Это сходство будет использоваться вместе с персонализацией, обучением ранжированию и графами знаний для надежного поиска. Поиск по плотным векторам на основе ближайшего соседа быстро набирает популярность и, вероятно, в какой-то момент вытеснит булево сопоставление с ранжированием BM25 в качестве наиболее распространенного метода поиска и ранжирования для поиска неструктурированного текста. Однако два подхода – поиск по плотным векторам и лексический поиск – дополняют друг друга, а гибридные подходы, объединяющие оба, обычно работают даже лучше.

13.7. Квантизация и обучение представлениям для более эффективного векторного поиска

В последнем разделе мы представили две понятия, обычно используемые для ускорения плотного векторного поиска: поиск ANN и реранкинг. Концептуально ANN – это способ сократить количество вычислений сходства векторов, необходимых во время запроса, путем эффективного поиска и фильтрации топовых векторов, которые с наибольшей вероятностью будут похожи на вектор запроса. Поскольку

поиск ANN является приближением лучших результатов, обычно делают реранкинг этих верхних потенциальных результатов с использованием более точных (и вычислительно затратных) вычислений сходства векторов, чтобы получить отзыв и релевантность на уровне поиска без оптимизации ANN.

Время вычислений и объем памяти, необходимые для представления и выполнения вычислений сходства векторов, напрямую связаны с размером искомых векторов. В этом разделе мы познакомим вас с несколькими дополнительными методами повышения эффективности векторного поиска:

- скалярной квантизацией;
- бинарной квантизацией;
- квантизацией продукта;
- обучением представлению «матрешки».

Квантизация – это метод, используемый для уменьшения объема памяти и вычислительной сложности числовых представлений данных, таких как векторы эмбединга. В контексте эмбединга квантизация подразумевает сжатие этих векторов путем уменьшения числа битов, используемых для представления признаков вектора. Эмбединги обычно представляются как числа с плавающей точкой (float), которые по умолчанию имеют размер 32 бита (или 4 байта). Если типичный векторный эмбединг имеет 1024 измерения, это преобразуется в 1024×4 байт, или 4096 байт (4 Кб) на вектор. Если у вас есть большое количество векторов для хранения и поиска, это может быстро составить значительный объем памяти и вычислительных затрат.

Квантизация эмбедингов позволяет вам пожертвовать некоторым идеально небольшим объемом отзыва для значительного улучшения эффективности хранения и скорости запросов, что имеет решающее значение для крупномасштабных поисковых систем. Например, вы можете уменьшить использование памяти вектором 32-битных float (Float32) на 75 %, преобразовав каждый признак в 8-битное целое число (Int8). Сжатие отдельных числовых значений (скаляров) каждого измерения, подобное этому, известно как *скалярная квантизация*. Это часто можно сделать без существенного влияния на отзыв, и это может быть особенно полезно, когда у вас есть большое количество векторов для хранения и поиска. Вы даже можете квантовать каждую функцию до одного бита – метод, известный как *бинарная квантизация* – и по-прежнему поддерживать относительно высокий уровень отзыва, если это сочетается с шагом реранкинга с использованием вектора более высокой точности для топ- N результатов. Рисунок 13.9 наглядно демонстрирует понятия скалярной и бинарной квантизации с использованием вектора, содержащего изображение обложки оригинала этой книги (предположим, что размеры вектора представляют пиксели на изображении).



Рис. 13.9. Квантизация данных: полная точность, уменьшенная скалярная точность и бинарная точность

На рисунке вы можете видеть исходное изображение (без квантизации), скалярное квантованное изображение (использующее уменьшенную цветовую палитру, которая сопоставляется с аналогичными диапазонами, но с меньшим количеством оттенков серого цвета и, соответственно, меньшей точностью) и бинарное квантованное изображение (двухцветное изображение, где каждый пиксель либо черный, либо белый). Вы заметите, что скалярное квантованное изображение по-прежнему сохраняет большую часть важных деталей исходного изображения, а бинарная квантованная версия по-прежнему четко распознается, хотя и теряет некоторые важные данные (часть букв названия и оттенки серого цвета).

В этом разделе мы рассмотрим скалярную квантизацию, бинарную квантизацию и третий тип квантизации, называемый квантизацией произведения. Мы также представим многоуровневый подход к внедрению, известный как «матрешка» (Matryoshka Representation Learning), который можно использовать для динамического переключения уровней точности уже сгенерированных эмбедингов без необходимости дополнительной квантизации или переобучения.

13.7.1. Скалярная квантизация

Скалярная квантизация – это простейшая форма квантизации, где каждое значение в векторе эмбединга независимо преобразуется в представление с более низкой точностью. Рассмотрим следующие два вектора:

```
[ -1.2345679, 2.2345679, 100.45679 ] #4 bytes = 32 bits
[ -1.234, 2.234, 100.44 ] #2 bytes = 16 bits
```

Первый вектор – это 32-битное представление с плавающей точкой, а второй – 16-битное представление с плавающей точкой, округленное

максимально до приемлемой точности. Второй вектор требует на 50 % меньше памяти (2 байта против 4 байт), при этом представляя приблизительно те же значения, просто с несколько меньшей точностью.

Эта пониженная точность – простой пример скалярной квантизации, берущей значения с высокой точностью и преобразующей их в представления с меньшей точностью, требующие меньше памяти для хранения и меньше вычислительных затрат при обработке.

Но что, если мы хотим сжать наши векторы еще больше, до одного байта (или даже нескольких бит) – можем ли мы сделать это, сохранив большую часть нашего отзыва? Ответ – да, и мы обычно достигаем этого, преобразуя диапазон значений величин с плавающей точкой в Int8, как показано на рис. 13.10.

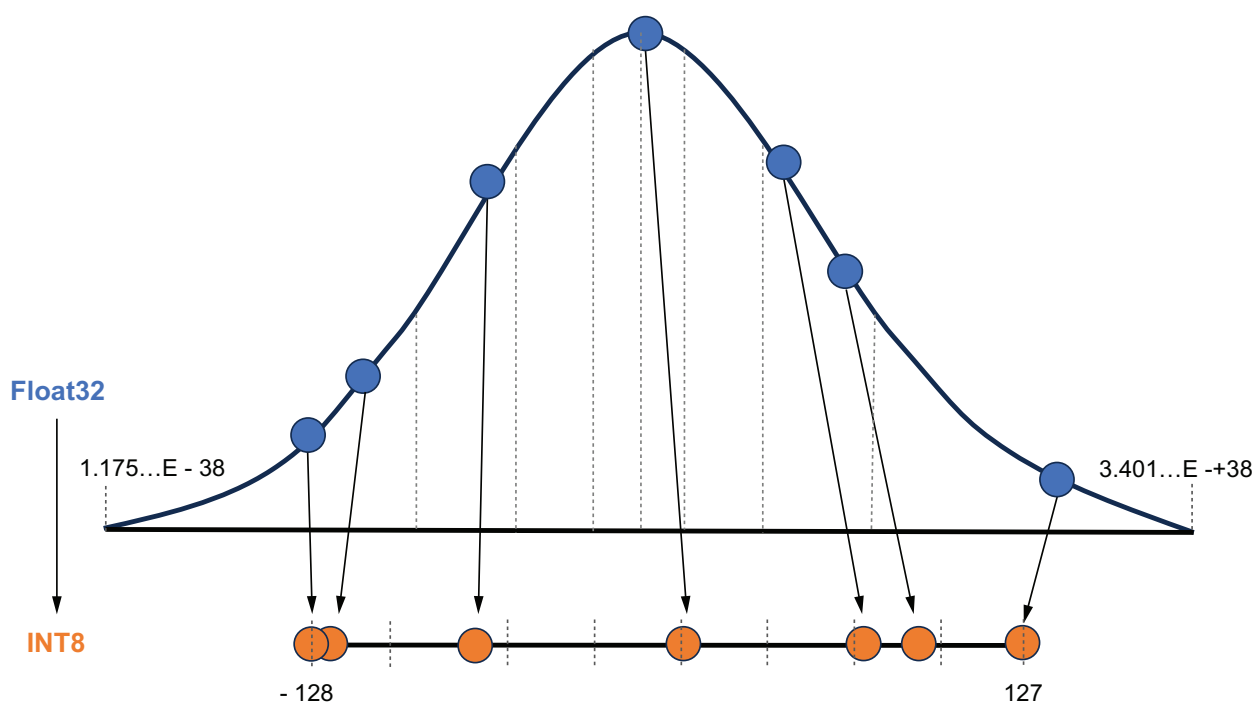


Рис. 13.10. Скалярная квантизация из Float32 в Int8

На рисунке кривая сверху представляет распределение значений в диапазоне с плавающей точкой. Поскольку мы квантуем из 32 до 8 бит, мы отображаем диапазон значений Float32 в меньший диапазон от -128 до 127 (или от 0 до 255, если используется беззнаковое целое число). В зависимости от используемого алгоритма квантизации часто предпринимаются попытки использовать новые диапазоны как можно полнее, *ограничивая* значения (ограничивая диапазон минимумом и максимумом), а также используя плотность значений в исходном векторе для более равномерного преобразования в новый квантованный диапазон.

Давайте реализуем пример скалярной квантизации, чтобы увидеть, какой эффект эта оптимизация дает на размер индекса, отзыв и скорость поиска. Существует несколько библиотек для выполнения скалярной квантизации. Вы можете использовать модуль `sentence_transformers.quantization`, или ваша поисковая система или языковая модель

могут иметь свои собственные встроенные реализации квантизации. Мы собираемся использовать библиотеку FAISS для наших квантованных индексов и комбинацию для библиотеки `sentence_transformers` и FAISS для каждого из наших примеров квантизации. FAISS (Facebook AI Similarity Search) – это библиотека с открытым исходным кодом, разработанная для эффективного поиска по сходству и кластеризации плотных векторов. Она похожа на библиотеку NMSLIB, которую мы использовали ранее в этой главе для семантического поиска, но имеет некоторые дополнительные функции, включая встроенную поддержку квантизации. FAISS широко используется в производственных системах для плотного поиска векторов и является отличным выбором для реализации квантованных индексов.

Позже мы рассмотрим пример того, как запустить поиск с использованием абстракции `collection` для выбранной вами поисковой системы, но, поскольку не каждая система поддерживает все режимы квантизации и поскольку каждая система имеет разную затратность и разную производительность, для нашего сравнительного анализа мы будем использовать FAISS.

Теперь давайте создадим индекс FAISS с эмбедингами Float32 полной точности. Затем мы будем использовать его в качестве нашей базовой линии по сравнению с различными квантованными индексами эмбединга для сравнения размеров индексов, скоростей поиска и показателей отзыва. В листинге 13.20 показан код для создания эмбедингов Float32 полной точности и их индексации в индексе FAISS.

Листинг 13.20. Индексирование эмбедингов полной точности с использованием FAISS

```
from sentence_transformers.quantization import quantize_embeddings

model = SentenceTransformer(
    "mixedbread-ai/mxbai-embed-large-v1",
    similarity_fn_name=SimilarityFunction.DOT_PRODUCT,
    truncate_dim=1024)

def index_full_precision_embeddings(doc_embeddings, name):
    index = faiss.IndexFlatIP(doc_embeddings.shape[1])
    index.add(doc_embeddings)
    faiss.write_index(index, name)
    return index

def get_outdoors_embeddings(model):
    outdoors_dataframe = load_dataframe("data/outdoors/posts.csv")
    post_texts = [post["title"] + " " + post["body"]
                  for post in outdoors_dataframe.collect()]
    return numpy.array(
        get_embeddings(post_texts, model, "outdoors_mrl_normed"))
```

Эта модель создает эмбединги, поддерживающие все будущие методы оптимизации.

Записывает индекс на диск.

Исходные эмбединги будут иметь 1024 измерения.

Добавляет документы в индекс.

IndexFlatIP – это простой неоптимизированный индекс, поддерживающий различные форматы эмбедингов.

```
doc_embeddings = get_outdoors_embeddings(model)
full_index = index_full_precision_embeddings(
    doc_embeddings, "full_embeddings")
```

Генерирует эмбединги для набора данных outdoor.

Создает индекс FAISS полной точности (Float32).

В этом листинге мы вычисляем эмбединги для набора данных outdoors, используя модель `mixedbread-ai/mxbai-embed-large-v1`, которая создает высококачественные эмбединги, хорошо работающие со всеми нашими методами квантизации, а также поддерживает метод Matryoshka Representation Learning, который мы рассмотрим далее в этой главе. Затем мы индексируем эти эмбединги в индекс FAISS полной точности (Float32), который мы вскоре будем использовать в качестве базового уровня для сравнительного анализа производительности различных методов квантизации.

Для наших тестов нам также нужно будет закодировать некоторые тестовые запросы в эмбединги, что показано в листинге 13.21.

Листинг 13.21. Генерация эмбедингов запросов и полноиндексных бенчмарков

```
def get_test_queries():
    return ["tent poles", "hiking trails", "mountain forests",
            "white water", "best waterfalls", "mountain biking",
            "snowboarding slopes", "bungee jumping", "public parks"]

queries = get_test_queries()
query_embeddings = model.encode(queries,
                                convert_to_numpy=True,
                                normalize_embeddings=True)

full_results = time_and_execute_search(
    full_index, "full_embeddings",
    query_embeddings, k=25)
display_statistics(full_results)
```

Получает тестовые запросы для бенчмарка.

Генерирует эмбединги для каждого запроса.

Генерирует время поиска, размер индекса и статистику отклика для индекса полной точности (Float32).

Отображает статистику бенчмарка.

Вывод:

```
full_embeddings search took: 7.621 ms
full_embeddings index size: 75.6 MB
Recall: 1.0
```

В предыдущем листинге мы определяем список запросов, которые будем использовать для сравнительного анализа нашего индекса полной точности с различными стратегиями оптимизации векторного поиска. Затем мы вызываем функцию `time_and_execute_search` (опущена для краткости) для индекса полной точности из листинга 13.20, а затем передаем результаты в функцию `display_statistics` (также опущена для краткости), которая отображает время поиска, размер индекса и статистику отклика.

Это обеспечивает базовую линию для сравнения с нашими будущими квантованными (или иным образом оптимизированными) индек-

сами. Листинг 13.22 показывает реализацию двух дополнительных функций, которые мы будем использовать для сравнения результатов других стратегий индексирования: функции `evaluate_search` и `evaluate_rerank_search`.

Листинг 13.22. Функции бенчмарка оптимизированных подходов к поиску

```
def evaluate_search(full_index, optimized_index,
                   optimized_index_name,
                   query_embeddings,
                   optimized_query_embeddings,
                   k=25, display=True, log=False):
    full_results = time_and_execute_search(
        full_index, "full_embeddings",
        query_embeddings, k=k)
    optimized_results = time_and_execute_search(
        optimized_index,
        optimized_index_name,
        optimized_query_embeddings, k=k)

    if display:
        display_statistics(optimized_results, full_results)
    return optimized_results, full_results
```

Возвращает 25 лучших результатов из каждого индекса и сравнивает их.

Вычисляет скорость запроса, размер индекса и отзыв для оптимизированного индекса по сравнению с индексом полной точности.

```
def evaluate_rerank_search(full_index, optimized_index,
                           query_embeddings,
                           optimized_embeddings,
                           k=50, limit=25):
    results, full_results = evaluate_search(
        full_index,
        optimized_index, None,
        query_embeddings,
        optimized_embeddings,
        display=False, k=k)
```

То же, что и `estimate_search`, но перезапрашивает результаты `k=50` (по умолчанию) и переранжирует те, которые используют эмбединги полной точности.

```
doc_embeddings = get_outdoors_embeddings(model)
rescore_scores, rescore_ids = [], []
for i in range(len(results["results"])):
    embedding_ids = results["faiss_ids"][i]
    top_k_embeddings = [doc_embeddings[id]
                        for id in embedding_ids]
    query_embedding = query_embeddings[i]
    scores = query_embedding @ \
        numpy.array(top_k_embeddings).T
    indices = scores.argsort()[::-1][:limit]
    top_k_indices = embedding_ids[indices]
    top_k_scores = scores[indices]
    rescore_scores.append(top_k_scores)
    rescore_ids.append(top_k_indices)
```

Генерирует эмбединги для каждого документа в наборе данных `outdoors`.

Выполняет скалярное произведение между эмбедингом запроса и эмбедингами топ-к.

Сортирует результаты по баллу скалярного произведения, чтобы сделать их реранкинг.

```

results = generate_search_results(rescore_scores, rescore_ids)
recall = calculate_recall(full_results["results"], results)
print(f"Reranked recall: {recall}")

```

Вычисляет отзыв результатов после реранкинга по сравнению с индексом полной точности.

Вспомогательная функция (опущена), которая объединяет идентификаторы и баллы с другими полями документа для возврата.

Функция `evaluate_search` внутренне вызывает функцию `time_and_execute_search` как для индекса полной точности, так и для квантованного индекса и передает результаты в функцию `display_statistics` для сравнения и отображения времени поиска, размера индекса и статистики отзыва.

Затем функция `evaluate_rerank_search` снова вычисляет отзыв после реранкинга топ- N результатов из квантованного индекса с использованием эмбедингов полной точности. Хотя квантизация может радикально сократить память и время поиска, мы также увидим, что она уменьшает отзыв, т. е. некоторые из результатов, которые *должны* быть возвращены, не возвращаются. Но, запрашивая и повторно ранжируя только высшие N результатов с использованием эмбедингов полной точности (обычно загружаемых с диска после квантованного поиска, а не хранящихся в памяти вместе с индексом), мы можем восстановить большую часть этого потерянного отзыва.

Мы покажем как квантованный отзыв, так и переранжированный квантованный отзыв в нескольких последующих листингах, чтобы продемонстрировать компромиссы между поисками полной точности, квантованными поисками и переранжированными квантованными поисками. Для нашего первого примера квантизации давайте реализуем скалярную квантизацию `Int8`.

Листинг 13.23. Создание квантованного индекса эмбедингов `Int8` с использованием `FAISS`

```

def index_int8_embeddings(doc_embeddings, name):
    int8_embeddings = quantize_embeddings(
        doc_embeddings, precision="int8")
    print("Int8 embeddings shape:", int8_embeddings.shape)
    index = faiss.IndexFlatIP(int8_embeddings.shape[1])
    index.add(int8_embeddings)
    faiss.write_index(index, name)
    return index

int8_index_name = "int8_embeddings"
int8_index = index_int8_embeddings(doc_embeddings, int8_index_name)

quantized_queries = quantize_embeddings(
    query_embeddings,
    calibration_embeddings=doc_embeddings,
    precision="int8")

```

Квантует эмбединги документа до точности `Int8`.

Создает индекс, настроенный на ожидание формы эмбедингов.

Добавляет квантованные эмбединги в индекс.

Сохраняет индекс на диск, чтобы мы могли измерить его размер.

Квантует эмбединги запроса до точности `Int8`.


```
evaluate_search(full_index, int8_index,  
                int8_index_name, query_embeddings,  
                quantized_queries)  
evaluate_rerank_search(full_index, int8_index,  
                      query_embeddings, quantized_queries)
```

Выполняет бенчмарки для времени поиска, размера индекса и отзыва.

Снова выполняет бенчмарки, позволяя переранжировать лучшие результаты с эмбедингами полной точности.

Вывод:

```
Int8 embeddings shape: (18456, 1024)  
int8_embeddings search took: 9.070 ms (38.65% improvement)  
int8_embeddings index size: 18.91 MB (74.99% improvement)  
Recall: 0.9289  
Reranked recall: 1.0
```

В выводе листинга 13.23 мы видим, что квантованный индекс Int8 на 75 % меньше индекса полной точности, что имеет смысл, учитывая, что мы уменьшили точность с Float32 до Int8, что составляет ~75 % сокращения с 32 до 8 бит. Аналогично мы видим, что общее время поиска улучшилось из-за того, что числа с меньшей точностью стали более эффективными для обработки (по крайней мере, в некоторых системах). Обратите внимание, что скорость поиска будет *значительно* различаться во всех этих тестах от системы к системе и от запуска к запуску, но размер индекса всегда должен быть одинаковым. Также обратите внимание, что в нашем индексе довольно небольшое количество документов, и улучшения скорости запросов от квантизации, вероятно, станут более выраженными по мере увеличения количества документов.

Однако наиболее важными числами являются числа отзыва. Квантованный поиск Int8 сохранил уровень отзыва 92.89 %. Это означает, что мы достигли ~75 % сокращения размера индекса и значительного улучшения скорости поиска, при этом всего 7.11 % лучших результатов $N=25$ в квантованном поиске были пропущены. Так же как среднее изображение обложки этой книги на рис. 13.9 сохранило подавляющее большинство важных деталей исходного изображения, мы смогли сохранить высокую точность наших квантованных эмбедингов в листинге 13.23 при использовании квантизации Int8.

Значение Reranked recall 1.0 дополнительно указывает на то, что, просто запросив топ- $N=50$ результатов и переранжировав их с использованием исходных эмбедингов полной точности, мы возвращаемся к 100%-му отзыву. Это распространенная схема при использовании квантизации в плотном векторном поиске: выполните первоначальный поиск, который запрашивает избыточные результаты с использованием квантованного индекса (для значительного улучшения памяти и скорости), а затем сделаете реранкинг топ- N результатов с использованием эмбедингов с более высокой точностью (обычно извлекаемых с диска, чтобы они не влияли на ваши требования к индексу и памяти), чтобы восстановить потерянный отзыв и точ-

ность ранжирования. Хотя эти улучшения впечатляют, мы могли бы продолжить сжатие еще больше, используя 4 бита (Int4) или меньше. В следующем разделе мы увидим, что произойдет, если мы сожмем каждое измерение до одного бита!

13.7.2. Бинарная квантизация

Бинарная квантизация – это экстремальная форма квантизации, в которой каждое значение в векторе эмбединга представлено одним битом, что сводит значения к 0 или 1. Этот метод похож на преобразование изображения в черно-белый цвет (только один оттенок черного, без оттенков серого), как в примере справа на рис. 13.9.

Квантизацию каждого признака в один бит можно выполнить с помощью простого порога для назначения бита 0 для любого признака, меньшего или равного 0, и значения 1 для любого признака, большего 1.0. Это хорошо работает, если значения признаков в эмбедингах имеют равномерное распределение между положительными и отрицательными значениями. Однако, если значения признаков неравномерно распределены по документам, может быть полезно использовать медианное значение для каждого признака или какой-либо аналогичный порог для назначения 0 и 1 в качестве бинарного квантованного значения, чтобы более равномерно распределить значения.

Рисунок 13.11 иллюстрирует результат бинарной квантизации, где сохраняется только самая важная информация сродни черно-белой версии обложки этой книги. Крайнее левое изображение демонстрирует использование простого порога, среднее изображение представляет использование неравномерного порога, который назначает значения на основе относительного распределения значений для каждого признака в исходных эмбедингах, а крайнее правое изображение показывает оптимальное обученное бинарное представление непосредственно из модели трансформер-кодировщика. Это последняя «основанная на модели» бинарная квантизация выглядит лучше всего, потому что выбор значений учитывает контекст всего изображения при генерации бинарного представления, а не только отдельных признаков. Это позволяет гораздо более осмысленно и точно представить исходный эмбединг, закодированный в доступные бинарные квантованные значения признаков на основе того, как модель была обучена понимать изображение в целом.

Мы не будем демонстрировать пример выполнения бинарной квантизации на основе модели, поскольку пытаемся сравнить отзыв для тех же эмбедингов на разных уровнях квантизации, но стоит помнить, что, если ваша модель поддерживает бинарную квантизацию на основе модели, она, как правило, даст лучшие результаты, чем выполнение бинарной квантизации на признаках после того, как модель уже закодировала и вернула их с более высоким уровнем точности.

Давайте теперь реализуем бинарную квантизацию с использованием FAISS и сравним результаты.



Рис. 13.11. Различные методы бинарной квантизации

Листинг 13.24. Индексирование и бенчмарк бинарных квантованных эмбеддингов

```
def index_binary_embeddings(doc_embeddings,
                           binary_index_name):
    binary_embeddings = quantize_embeddings(
        doc_embeddings,
        precision="binary").astype(numpy.uint8)
    print("Binary embeddings shape:", binary_embeddings.shape)
    index = faiss.IndexBinaryFlat(
        binary_embeddings.shape[1] * 8)
    index.add(binary_embeddings)
    faiss.write_index_binary(index, binary_index_name)
    return index

binary_index_name = "binary_embeddings"
binary_index = index_binary_embeddings(
    doc_embeddings, binary_index_name)

quantized_queries = quantize_embeddings(
    query_embeddings,
    calibration_embeddings=doc_embeddings,
    precision="binary").astype(numpy.uint8)

evaluate_search(full_index, binary_index,
                binary_index_name,
                query_embeddings, quantized_queries)
evaluate_rerank_search(full_index, binary_index,
                       query_embeddings, quantized_queries)
```

Квантует эмбеддинги документа в бинарный код (1 бит на измерение).

Создает индекс бинарных эмбеддингов.

Добавляет все эмбеддинги документа в индекс.

Записывает индекс на диск.

Квантует эмбеддинги запроса в бинарный код.

Эмбеддинги документа предоставляются квантователю для калибровки наилучших пороговых значений для назначения 0 или 1.

Сохраняет каждые 8 измерений как 1 байт, закодированный как беззнаковый Int8.

Выполняет бенчмарки с реранкингом и без него по сравнению с индексом полной точности.

Вывод:

```
Binary embeddings shape: (18456, 128)
binary_embeddings search took: 1.232 ms (83.38% improvement)
binary_embeddings index size: 2.36 MB (96.87% improvement)
Recall: 0.6044
Reranked recall: 1.0
```

Хотя неудивительно, что бинарная квантизация уменьшает размер индекса и требования к памяти на 96.87 % (с 32 до 1 бита), ошеломляет то, что нам удалось сохранить 60.44 % отклика всего с одним битом на признак эмбединга. Хотя 60 % отклика не обязательно достаточно для многих случаев использования поиска, обратите внимание, что, когда мы перезапросили в 2 раза ($N=50$, чтобы найти 25 лучших) и сделали реранкинг, мы смогли вернуть для наших тестовых запросов отклик до 100 %. В более крупном наборе данных и количестве запросов вы, скорее всего, не сможете поддерживать такой отклик, но даже приближение к отклику 100 % всего с одним битом на признак для вашего первоначального поиска является впечатляющим достижением.

С возможностью достижения такого экстремального сжатия, при этом имея возможность получить почти 100%-ный отклик путем перезапроса и реранкинга, бинарная квантизация расширяет границы того, что дает просто квантизация. Вы можете подумать, что один бит на измерение будет пределом того, насколько глубоко мы можем квантовать наши эмбединги, но в следующем разделе мы представим метод, который позволяет нам делать еще более сильное сжатие.

13.7.3. Продуктовая квантизация

В то время как скалярная и бинарная квантизации фокусируются на снижении точности отдельных признаков эмбединга, *продуктовая квантизация* (PQ) вместо этого фокусируется на квантовании всего эмбединга до более эффективного с точки зрения памяти. Она обеспечивает еще более глубокое сжатие, чем бинарная квантизация, что делает PQ особенно полезной для крупномасштабных поисковых приложений с большим количеством измерений эмбединга.

Представьте, что у вас есть длинное предложение, и вы разбиваете его на более короткие фразы; проще обрабатывать и обрабатывать каждую фразу независимо. Аналогично PQ делит векторное пространство на меньшие области (подвекторы) и квантует каждый подвектор по отдельности. Затем PQ кластеризует каждую из областей подвектора, чтобы найти набор центроидов кластера, и наконец назначает один идентификатор центроида каждому подвектору для каждого документа. Этот список идентификаторов центроидов (по одному на подвектор) для каждого документа называется PQ-кодом, и это квантованное представление эмбединга документа. Рисунок 13.12 демонстрирует процесс PQ.

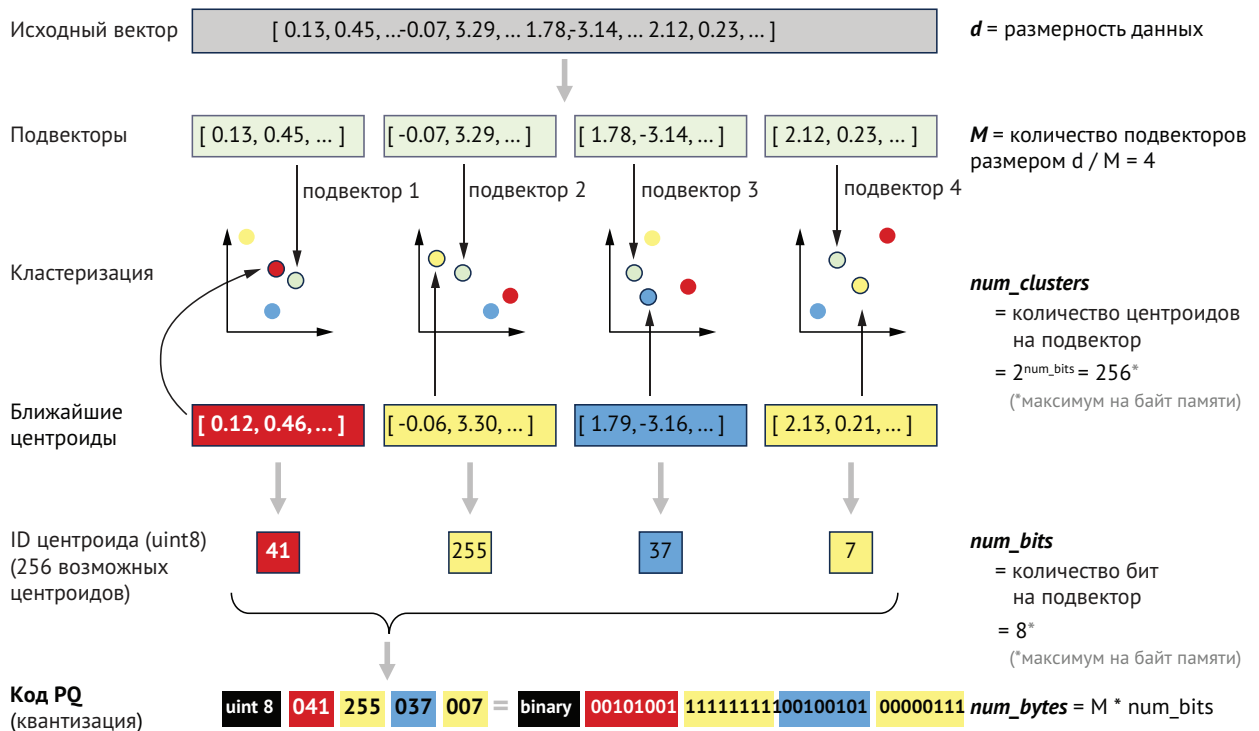


Рис. 13.12. Процесс продуктовой квантизации

Процесс квантизации начинается с верхнего слоя рисунка и последовательно продвигается вниз. Сначала исходное векторное пространство с d измерениями делится на M подвекторных пространств. Для каждого из M подвекторных пространств соответствующий подвектор разделяется из каждого исходного эмбединга. Затем выполняется кластеризация (обычно с использованием алгоритма k -средних) для всех подвекторов в каждом подпространстве для создания списка кластеров $num_clusters$, идентифицированных по их вектору центра в подпространстве. Затем каждому центру назначается ID, который записывается в *кодую книгу*, содержащую сопоставление каждого идентификатора центра с его полными подвекторами. Наконец, идентификаторы центра для каждого подпространства объединяются вместе для каждого документа для создания PQ-кода, который является официальным квантованным представлением каждого документа.

Затем PQ-коды документов индексируются, и во время запроса эмбединг запроса делится на те же M подпространств, а расстояние между подвектором запроса в этом подпространстве и каждым центром в подпространстве вычисляется и кешируется в таблице поиска. Поскольку PQ-код каждого документа сопоставляется с определенным центром в каждом подпространстве, мы можем получить приблизительное расстояние между подвектором запроса и подвектором документа, найдя его в кешированной таблице поиска. Далее оценки сходства векторов между запросом и каждым документом можно вычислить и ранжировать по наименьшему расстоянию в объединенных подпространствах, используя метрику, такую как внутренний продукт или евклидово расстояние. Например, для вычисления евклидова расстояния требуется просто извлечь квадрат-

ный корень из суммы квадратов расстояний от каждого подвектора. Такие библиотеки, как FAISS, предоставляют эффективные готовые реализации PQ. Листинг 13.25 демонстрирует процесс построения индекса с использованием PQ.

Листинг 13.25. Создание и тестирование индекса квантизации продукта

Делит эмбединг на $M=16$ подвекторов (по 64 измерения каждый).

Исходный эмбединг имеет 1024 измерения.

```
def index_pq_embeddings(doc_embeddings, index_name, num_subvectors=16):
```

```
    dimensions = doc_embeddings.shape[1]
```

```
    M = num_subvectors
```

```
    num_bits = 8
```

```
    index = faiss.IndexPQ(dimensions, M, num_bits)
```

```
    index.train(doc_embeddings)
```

```
    index.add(doc_embeddings)
```

```
    faiss.write_index(index, index_name)
```

```
    return index
```

8 бит = 256 максимальных центроидов кластера на подвектор.

Создает индекс PQ.

Генерирует центроиды кластера с использованием кластеризации k-средних.

Сохраняет индекс на диск, чтобы мы могли измерить размер.

Добавляет все doc_embeddings в индекс.

```
pq_index_name = "pq_embeddings"
```

```
pq_index = index_pq_embeddings(doc_embeddings,
                               pq_index_name)
```

```
evaluate_search(full_index, pq_index, pq_index_name,
                query_embeddings, query_embeddings)
```

```
evaluate_rerank_search(full_index, pq_index,
                       query_embeddings,
                       query_embeddings)
```

Запускает бенчмарки по сравнению с индексом полной точности.

Вывод:

```
pq_embeddings search took: 2.092 ms (75.22% improvement)
```

```
pq_embeddings index size: 1.34 MB (98.22% improvement)
```

```
Recall: 0.3333
```

```
Reranked recall: 0.6800
```

Вы сразу заметите, что отзыв для индекса PQ значительно ниже, чем у скалярных и бинарных типов квантизации, которым мы делали бенчмаркинг, – на 33.33 % отзыва без реранкинга и 68 % отзыва после реранкинга. Однако вы также заметите, что размер индекса на 98.22 % меньше, чем у исходного индекса полной точности. Очевидно, мы намеренно пошли на компромисс, чтобы гипероптимизировать размер индекса в обмен на отзыв. Однако, в отличие от методов скалярной и бинарной квантизации, PQ предоставляет рычаги для корректировки этого компромисса путем увеличения либо количества подвекторов ($M_{subvectors}$), либо num_bits для хранения большей точности в индексе и улучшения отзыва. Например, если бы вы установили $M_{subvectors}$ на 64, вы бы получили следующее:


```
pq_embeddings search took: 4.061 ms (43.99% improvement)
pq_embeddings index size: 2.23 MB (97.05% improvement)
Recall: 0.5778
Reranked recall: 0.9911
```

Эти результаты теперь больше соответствуют результатам бинарной квантизации. Ключевым выводом, таким образом, является то, что основное преимущество PQ заключается в гибкости управления компромиссом между размером индекса и отзывом, особенно когда вам нужно значительно сжать ваши эмбединги.

Теперь мы изучили несколько подходов к квантизации, чтобы уменьшить размер индексированных эмбедингов и ускорить время поиска, сохраняя при этом очень высокий уровень отзыва относительно уровня сжатия. В следующем подразделе мы рассмотрим другой подход к решению компромисса между сжатием и отзывом, представив эмбединги, которые фактически кодируют несколько уровней точности внутри исходного представления эмбединга.

13.7.4. Репрезентативное обучение *Matryoshka*

Matryoshka Representation Learning (MRL) – это новый подход к оптимизации производительности векторов, который изучает иерархическое представление векторного пространства, где несколько уровней точности кодируются различными диапазонами измерений в векторном пространстве. Это позволяет использовать представления с гибкой длиной, где более короткие сегменты эмбединга могут аппроксимировать значение полного эмбединга, просто с уменьшенной точностью. MRL назван в честь русских матрешек, что указывает на слои точности, которые можно обнаружить внутри других слоев.

В качестве концептуального объяснения того, как работает MRL, представьте, что кто-то, кто никогда не видел анимационный фильм Disney «Король Лев», попросил вас описать его. Если вы знакомы с фильмом, можете начать с очень общего резюме, а затем расширить свое описание, если человек захочет больше подробностей. Например, вы можете представить следующие возможные ответы:

- 1 это анимационный фильм Disney о льве;
- 2 это анимационный фильм Disney о львенке, который вырастает и становится королем;
- 3 это анимационный фильм Disney о львенке по имени Симба, который вырастает и становится королем после того, как его отца убивает дядя;
- 4 это анимационный фильм Disney о львенке по имени Симба, который должен найти свое место в круге жизни после того, как сбежал из своего королевства мальчиком и вернулся взрослым, чтобы вернуть свой трон, отобрав его у дяди, который убил его отца.

Обратите внимание, что все они передают основную идею истории на разных уровнях детализации. Если бы было действительно важно предоставить наиболее точное описание, вы бы выбрали наиболее

подробное, но на самом деле в некоторых случаях может быть эффективнее начать с описания более высокого уровня и предоставлять дополнительные уровни детализации только по мере необходимости.

Эта концепция иерархических представлений, которая раскрывает больше о предмете на разных иерархических уровнях детализации, является ключевой идеей MRL. Эта техника позволяет векторам иметь прогрессивную точность, т. е. чем больше вектора вы используете, тем точнее становится представление. Чтобы проиллюстрировать это, давайте рассмотрим рис. 13.13, на котором показана обложка оригинала этой книги на разных уровнях пикселизации.

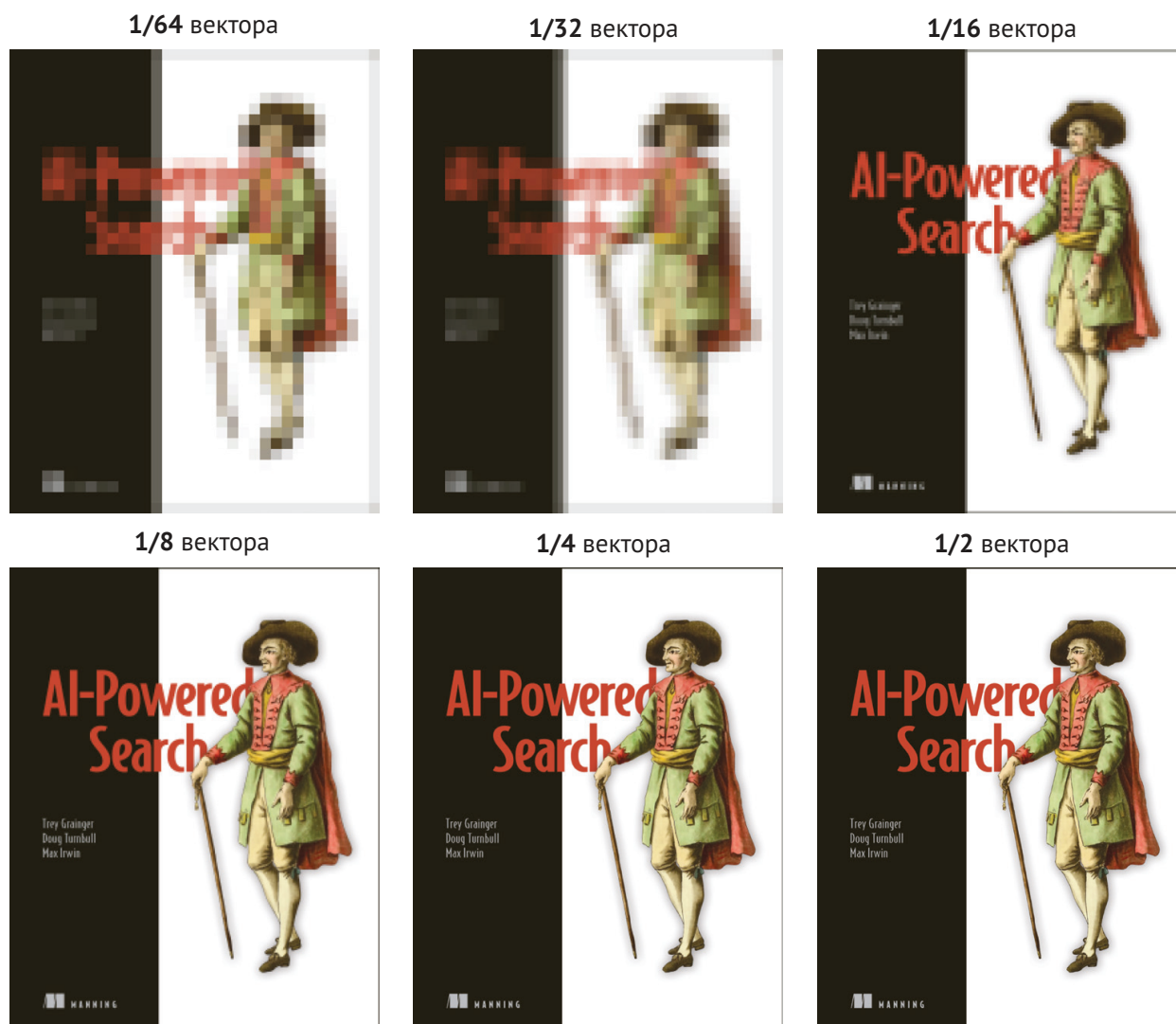


Рис. 13.13. Иерархические уровни представления, уравнивающие точность и сжатие

Как вы можете видеть, изображение в верхнем левом углу сильно пикселизировано, представляя собой очень грубое приближение исходной обложки. При движении вправо, а затем к следующему ряду каждое последующее изображение включает в себя вдвое больше измерений, в конечном итоге приводя к четкому и подробному представлению. Эти измерения являются *уточнениями* предыдущих измерений, однако они не являются полностью новой информацией. Дру-

гими словами, можно использовать полный вектор, только первую половину вектора, только первую четверть вектора и т. д. и все равно получить приближение исходного вектора, просто с меньшей точностью. Это похоже на то, как работают эмбединги MRL.

Листинг 13.26 демонстрирует построение и бенчмаркинг индекса FAISS с использованием эмбедингов MRL. Обратите внимание, что, поскольку представления с более низкой точностью достигаются путем простого отсечения более поздних измерений исходного эмбединга, для эмбедингов MRL не требуется никакой специальной стратегии индексации. Вам просто нужно уменьшить количество измерений плотного векторного поля, которое вы индексируете и ищете, до количества измерений для эмбедингов MRL, которые вы решили использовать (сокращая их пополам каждый раз и отбрасывая вторую половину).

Листинг 13.26. Тестирование эмбедингов MRL при разных пороговых значениях

```
def get_mrl_embeddings(embeddings, num_dimensions):
    mrl_embeddings = numpy.array(
        list(map(lambda e: e[:num_dimensions], embeddings)))
    return mrl_embeddings

def index_mrl_embeddings(doc_embeddings, num_dimensions, mrl_index_name):
    mrl_doc_embeddings = get_mrl_embeddings(doc_embeddings, num_dimensions)
    print(f"{mrl_index_name} embeddings shape:", mrl_doc_embeddings.shape)
    mrl_index = index_full_precision_embeddings(
        mrl_doc_embeddings, mrl_index_name)
    return mrl_index

print(f"Original embeddings shape:", doc_embeddings.shape)
original_dimensions = doc_embeddings.shape[1]

for num_dimensions in [original_dimensions//2,
                       original_dimensions//4,
                       original_dimensions//8]:

    mrl_index_name = f"mrl_embeddings_{num_dimensions}"
    mrl_index = index_mrl_embeddings(doc_embeddings,
                                    num_dimensions,
                                    mrl_index_name)
    mrl_queries = get_mrl_embeddings(query_embeddings,
                                    num_dimensions)

    evaluate_search(full_index, mrl_index, mrl_index_name,
                   query_embeddings, mrl_queries)
    evaluate_rerank_search(full_index,
                          mrl_index, query_embeddings, mrl_queries)
```

Используется только верхние измерения num_dimensions.

Индекс MRL – это стандартный индекс, просто с уменьшенным числом измерений.

1024 измерений.

512 измерений.

256 измерений.

128 измерений.

Сравнительный поиск MRL.

Сравнительный поиск MRL + реранкинг.

Вывод:

```
Original embeddings shape: (18456, 1024)
```

```
mrl_embeddings_512 embeddings shape: (18456, 512)
mrl_embeddings_512 search took: 3.586 ms (49.15% improvement)
mrl_embeddings_512 index size: 37.8 MB (50.0% improvement)
Recall: 0.7022
Reranked recall: 1.0
```

```
mrl_embeddings_256 embeddings shape: (18456, 256)
mrl_embeddings_256 search took: 1.845 ms (73.45% improvement)
mrl_embeddings_256 index size: 18.9 MB (75.0% improvement)
Recall: 0.4756
Reranked recall: 0.9689
```

```
mrl_embeddings_128 embeddings shape: (18456, 128)
mrl_embeddings_128 search took: 1.061 ms (84.35% improvement)
mrl_embeddings_128 index size: 9.45 MB (87.5% improvement)
Recall: 0.2489
Reranked recall: 0.64
```

В выходных данных мы видим, что 70.22 % отзыва было закодировано в первые 50 % признаков эмбединга, 47.56 % было закодировано в первые 25 % признаков, а 24.89 % было закодировано в первые 12.5 % признаков. Хотя этот уровень отзыва относительно сжатия может быть не таким впечатляющим, как некоторые из более ранних подходов квантизации, тот факт, что он встроен в представление эмбединга для использования (или неиспользования) по желанию без особых требований к индексации, обеспечивает некоторую полезную гибкость. Кроме того, как мы увидим в следующем подразделе, MRL также можно комбинировать с большинством других методов.

13.7.5. Объединение нескольких методов оптимизации векторного поиска

В этой главе мы обсудили несколько методов повышения эффективности векторного поиска:

- поиск ANN для фильтрации документов, которые должны быть оценены, до того количества, при котором большинство документов, скорее всего, будут оценены хорошо;
- методы квантизации для снижения требований к памяти и времени обработки для эмбедингов;
- MRL для сокращения количества измерений, необходимых для поиска начального набора результатов поиска;
- реранкинг для улучшения отзыва и ранжирования топ-N результатов после более агрессивного применения других методов.

На практике вы часто можете объединить несколько из этих методов для достижения желаемого уровня производительности. В листинге 13.27 показан последний пример, объединяющий каждый из этих подходов в одной реализации: ANN, использующая инвертиро-

ванный файловый индекс (IVF) для производительности, бинарная квантизация для производительности и сжатия, MRL с $1/2$ исходной размерности для производительности и сжатия и реранкинг в 2 раза большего количества результатов для улучшения отзыва и релевантности ранжирования.

Листинг 13.27. Объединение ANN, квантизации, MRL и реранкинга

```
def index_binary_ivf_mrl_embeddings(reduced_mrl_doc_embeddings,
                                   binary_index_name):
    binary_embeddings = quantize_embeddings(
        reduced_mrl_doc_embeddings,
        calibration_embeddings=reduced_mrl_doc_embeddings,
        precision="binary").astype(numpy.uint8)

    dimensions = reduced_mrl_doc_embeddings.shape[1]
    quantizer = faiss.IndexBinaryFlat(dimensions)

    num_clusters = 256
    index = faiss.IndexBinaryIVF(
        quantizer, dimensions, num_clusters)
    index.nprobe = 4

    index.train(binary_embeddings)
    index.add(binary_embeddings)
    faiss.write_index_binary(index, binary_index_name)
    return index

mrl_dimensions = doc_embeddings.shape[1] // 2
reduced_mrl_doc_embeddings = get_mrl_embeddings(
    doc_embeddings, mrl_dimensions)

binary_ivf_mrl_index_name = "binary_ivf_mrl_embeddings"
binary_ivf_mrl_index = index_binary_ivf_mrl_embeddings(
    reduced_mrl_doc_embeddings, mrl_dimensions,
    binary_ivf_mrl_index_name)

mrl_queries = get_mrl_embeddings(query_embeddings,
                                  mrl_dimensions)
quantized_queries = quantize_embeddings(mrl_queries,
    calibration_embeddings=reduced_mrl_doc_embeddings,
    precision="binary").astype(numpy.uint8)

evaluate_search(full_index, binary_ivf_mrl_index,
    binary_ivf_mrl_index_name,
    query_embeddings, quantized_queries)
evaluate_rerank_search(
    full_index, binary_ivf_mrl_index,
    query_embeddings, quantized_queries)
```

Бинарная квантизация: применяет квантизацию к doc embeddings.

Конфигурация, чтобы индекс знал, как doc embeddings были квантованы.

ANN: использует бинарно-квантованный индекс IVF для поиска ANN.

Обучает, добавляет документы и сохраняет объединенный индекс на диск.

MRL: получает эмбединги документов с уменьшенной размерностью.

MRL: получает эмбединги запросов с уменьшенной размерностью.

Бинарная квантизация: применяет квантизацию к эмбедингам запросов.

Делает бенчмарк бинарной ANN, бинарной квантизации и эмбедингов MRL.

Снова тестирует с реранкингом с использованием эмбедингов полной точности.

Вывод:


```
binary_ivf_mrl_embeddings search took: 0.064 ms (99.09% improvement)
binary_ivf_mrl_embeddings index size: 1.35 MB (98.22% improvement)
Recall: 0.3511
Reranked recall: 0.7244
```

Как видно из листинга, различные подходы к оптимизации эмбедингов можно комбинировать взаимодополняющими способами для достижения желаемого баланса между скоростью запроса, сжатием индекса и использованием памяти, а также окончательным рейтингом релевантности. В этом случае, объединив ANN, MRL, бинарную квантизацию и реранкинг, мы смогли достичь самого быстрого времени поиска (~99 % быстрее поиска с полной точностью) и самого маленького размера индекса (уменьшение ~98 %), при этом сохранив более 72 % отзыва после реранкинга.

Использование квантизации в поддерживаемом движке

Мы создали квантованные индексы с использованием FAISS, чтобы гарантировать, что все продемонстрированные подходы ANN и квантизации можно легко воспроизвести для размера индекса, скорости поиска и отзыва, независимо от настроенного вами engine по умолчанию. Различные поисковые системы имеют разные уровни поддержки подходов квантизации, некоторые выполняют квантизацию внутри движка, а некоторые требуют, чтобы вы выполняли квантизацию вне движка и настраивали соответствующий формат квантизации (FLOAT32, INT8, BINARY и т. д.). Метод поиска в нашем интерфейсе collection реализует поддержку скалярной квантизации и бинарной квантизации с использованием последнего подхода, принимая параметр `quantization_size` (ранее продемонстрированный в листинге 11.19), а MRL должен поддерживаться любым движком с возможностями векторного поиска путем усечения эмбедингов MRL перед индексацией и поиском. Реранкинг также поддерживается путем добавления раздела `rerank_query` к вашему поисковому запросу. Например:

```
{
  'query': [0, ..., 1],
  'query_fields': ['binary_embedding'],
  'quantization_size': 'BINARY',
  'order_by': [('score', 'desc')],
  'limit': 25,
  'rerank_query': {
    'query': [-0.01628, ..., 0.02110],
    'query_fields': ['full_embedding'],
    'quantization_size': 'FLOAT32',
    'order_by': [('score', 'desc')],
    'rerank_count': 50
  }
}
```


В этом примере реализуется начальный бинарный квантованный запрос (в той степени, в которой его поддерживает ваш движок) и возвращаются 25 лучших результатов после реранкинга 50 лучших результатов с использованием эмбедингов полной точности (Float32).

Объединение различных методов ANN, скалярной квантизации, бинарной квантизации, квантизации произведений и MRL может значительно повысить эффективность вашей системы векторного поиска. Существует множество других методов квантизации, и постоянно разрабатываются новые, но этот обзор должен дать вам отличную отправную точку, если вы хотите оптимизировать использование векторного поиска. Экспериментируя с этими методами и комбинируя их различными способами, вы можете оптимизировать свою поисковую систему для достижения желаемого баланса между скоростью, использованием памяти и отзывом, особенно при применении реранкинга в качестве последнего шага. Но повторное ранжирование на основе эмбедингов не всегда является лучшим способом получить наиболее релевантные окончательные результаты поиска. В следующем разделе мы рассмотрим часто лучший способ выполнить окончательный реранкинг лучших результатов: использование кросс-кодировщика.

13.8. Кросс-кодировщики и би-кодировщики – сравнение

В этой главе мы построили семантический поиск с использованием кодировщика на основе трансформеров для генерации эмбедингов. Наша стратегия включала поиск сходства между запросом и документом путем отдельного кодирования их обоих в эмбединги, а затем использования косинусного сходства для генерации оценки релевантности на основе сходства двух эмбедингов. Такой кодировщик, который отдельно кодирует каждый ввод в эмбединг, чтобы эти эмбединги можно было сравнивать, называется *би-кодировщиком*¹.

В отличие от би-кодировщика *кросс-кодировщик* кодирует пары входов вместе и возвращает оценку сходства. Предполагая, что вводы представляют собой запрос и документ, кросс-кодировщик объединит, а затем закодирует запрос и документ вместе, возвращая оценку сходства, измеряющую, насколько хорошо документ отвечает на запрос.

Функционально кросс-кодировщик все еще генерирует оценку сходства между двумя вводами, но он может захватывать общий контекст между запросом и документом таким образом, который би-кодировщик не может. Например, би-кодировщик может поместить запрос *mountain hike* («поход в горы») рядом с документом, содержа-

¹ Би-кодировщики подходят для сценариев, где важны скорость и масштабируемость, но при этом они менее точны, чем кросс-кодировщики. – *Прим. ред.*

щим текст «first time snow hiking» («первый поход по снегу»), поскольку они оба связаны с похожим понятием – походы. Но кросс-кодировщик передаст и запрос, и документ кодировщику (трансформеру), который затем может определить с помощью своего механизма внимания, что, хотя оба ввода о походах, документ о походах в горы для начинающих (которые, вероятно, не будут включать в себя поход в горы в первый раз), а не конкретно о запросе похода в горы. Используя контекст запроса для интерпретации документа, кросс-кодировщик может, таким образом, достичь более тонкой интерпретации того, насколько хорошо документ соответствует запросу, чем би-кодировщик, который интерпретирует запрос и документ только независимо.

Рисунок 13.14 визуализирует архитектуры би-кодировщика и кросс-кодировщика.

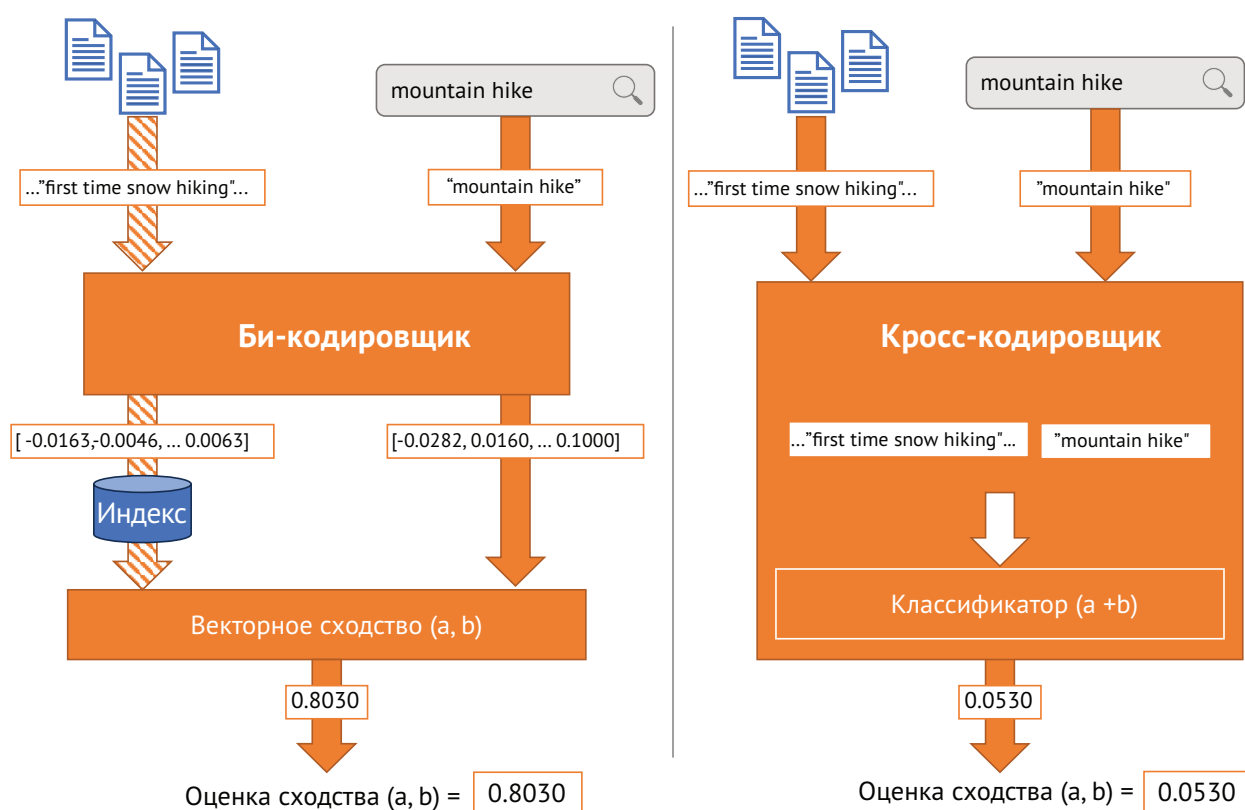


Рис. 13.14. Сравнение би-кодировщиков с кросс-кодировщиками. Би-кодировщики обрабатывают запросы и входные данные документа по отдельности, тогда как кросс-кодировщики обрабатывают их вместе, в общую оценку сходства

На рисунке обратите внимание на следующие ключевые признаки:

- би-кодировщики кодируют запросы и документы по отдельности в эмбединги, которые затем можно сравнивать с помощью функции сходства (например, косинусного сходства);
- кросс-кодировщики кодируют запросы вместе, чтобы назначить оценку сходства;
- сплошные стрелки указывают на обработку данных, которая должна выполняться во время запроса, тогда как полосатые стрелки указывают на работу, которая может быть выполнена во время индекси-

рования и кеширована. Обратите внимание, что би-кодировщикам нужно кодировать запрос только один раз во время запроса, тогда как кросс-кодировщики должны кодировать запрос вместе с каждым документом, которому требуется оценка сходства во время запроса.

Кросс-кодировщики во время запроса более затратны в вычислительном отношении, чем би-кодировщики, но они также обычно более точны, чем би-кодировщики, потому что они могут улавливать соответствующие взаимодействия между запросом и контекстами документа. По этой причине кросс-кодировщики часто используются для реранкинга небольшого подмножества лучших результатов из гораздо более быстрого векторного поиска на основе би-кодировщика или лексического поиска, чтобы обеспечить более точную оценку сходства для лучших пар запроса и документа.

Так же как и модели обучения ранжированию, которые мы построили в главах 10–12, кросс-кодировщики являются еще одной формой *классификатора ранжирования* – моделью, которая классифицирует входные данные по вероятностным оценкам сходства. Листинг 13.28 демонстрирует, как вызвать кросс-кодировщик для реранкинга результатов поиска. Мы будем использовать тот же начальный запрос из листинга 13.19, но запросим избыточное количество результатов поиска (`limit=50`), чтобы предоставить больше возможностей для реранкинга кросс-кодировщиком.

Листинг 13.28. Реранкинг результатов поиска с помощью кросс-кодировщика

<pre> from sentence_transformers import CrossEncoder cross_encoder = \ CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2") query = "mountain hike" search_results = semantic_search_with_engine(embeddings_collection, query, limit=50) pairs_to_score = [[query, doc["title"]] for doc in search_results] cross_scores = cross_encoder.predict(pairs_to_score, activation_fct=torch.nn.Sigmoid()) reranked_results = rerank(search_results, cross_scores) display_results(query, reranked_results[:10]) </pre>	Генерирует пару запрос + название документа для оценки каждого документа.	Модель кросс-кодировщика, обученная на похожем наборе данных вопрос–ответ.
	Запрашивает до 50 результатов сверх нормы, чтобы предоставить достаточно кандидатов для реранкинга.	
→ Вызывает кросс-кодировщик для оценки каждой пары.	Обновляет рейтинг релевантности на основе оценок кросс-кодировщика.	← Необязательная функция активации для нормирования между 0 и 1.

Ответ:

```

0.578 | What constitutes mountain exposure when hiking or scrambling?
0.337 | ... hiking trails... in... Rocky Mountain National Park...
0.317 | Where in the US can I find green mountains to hike...?
0.213 | Appropriate terms... hiking, trekking, mountaineering...

```

0.104 | Camping on top of a mountain
0.102 | ... Plan a Hiking Trip to Rocky Mountain National Park, CO
0.093 | What considerations... for... a hiking ascent of Mount...
0.073 | Are there any easy hiking daytrips up mountains...
0.053 | First time snow hiking
0.049 | Advice for first Grand Canyon Hike for Eastern Hikers

Хотя оценки не являются косинусным сходством (и, следовательно, не сопоставимы напрямую с оценками би-кодировщика), качество результатов после применения би-кодировщика, как правило, улучшается. Обратите внимание, что теперь больше документов связано с горным туризмом, а не с походами в целом.

В целом, глядя на эту главу, можно вывести один общий шаблон для интеграции кросс-кодировщиков с би-кодировщиками, который выглядит следующим образом:

- 1 выполните начальный поиск, используя комбинацию ИНС, квантизацию и методы обучения представлению (сверхбыстро);
- 2 переранжируйте среднее количество результатов (сотни или тысячи), используя N векторов более высокой точности, загруженных с диска, только для топ- N результатов (быстро, так как количество результатов сокращается);
- 3 возьмите одну или две верхние страницы результатов и переранжируйте их с помощью кросс-кодировщика, чтобы получить оптимальный рейтинг для верхних результатов (медленно, но только для небольшого количества результатов).

Конечно, вы можете использовать любую модель обучения ранжированию для вашего последнего шага ранжирования. Кросс-кодировщики являются одним из наиболее используемых видов модели обучения ранжированию (ориентированной исключительно на содержимое документа), поскольку они не требуют явно смоделированных признаков и вместо этого научились автоматически моделировать языковые и контентные признаки из процесса глубокого обучения. Вы можете (и должны) точно настроить свою модель кросс-кодировщика, используя методы из глав 10 и 11, но многие люди просто используют готовые предварительно обученные модели кросс-кодировщика без их тонкой настройки, поскольку они, как правило, хорошо обобщают при работе с общим текстовым контентом.

С тем, что вы узнали в этой главе, вы теперь должны уметь делать следующее с вашим собственным контентом:

- оценивать и выбирать существующую точно настроенную модель трансформер-кодировщика, которая соответствует вашему варианту использования;
- кодировать важный текст из ваших документов и добавлять его в индекс эмбедингов для поиска ANN;
- создавать конвейер автозаполнения для приема запросов с простым текстом и быстрого возврата наиболее тесно связанных понятий;
- добавлять в свой продукт мощный семантический поиск с высоким отзывом;

- оптимизировать производительность вашей плотной векторной поисковой системы, комбинируя методы ANN, квантизации, MRL и реранкинга;
- ранжировать результаты с помощью би-кодировщика и делать реранкинг этих результатов поиска с помощью кросс-кодировщика, чтобы улучшить рейтинг релевантности поиска.

Технология, лежащая в основе плотного векторного поиска, все еще нуждается в улучшении, поскольку ранжирование по эмбедингам может быть интенсивным в вычислениях, а подходы ANN имеют нетривиальные компромиссы масштабирования. Разреженные векторы терминов, использующие инвертированный индекс, по-прежнему намного эффективнее и проще масштабируются. Но огромный прогресс в направлении производственной реализации этих методов плотного векторного поиска продолжается, и на то есть веские причины. Поиск по векторам не только обеспечивает лучший семантический поиск по тексту, но и позволяет применять передовые подходы к ответам на вопросы, обобщению результатов, поиску изображений и другим вариантам использования расширенного поиска, таким как генерация дополненного поиска (RAG), – все это мы рассмотрим в следующих главах.

Резюме

- Поиск по плотным векторам ранжирует соответствующие документы, сравнивая расстояние между векторами эмбединга, например, из больших языковых моделей (LLM).
- Трансформеры позволяют LLM кодировать значение контента (запросов, документов, предложений и т. д.) в векторы, а также декодировать значение из закодированных векторов.
- Семантический поиск и другие варианты использования поиска, такие как семантическое автозаполнение, могут быть реализованы с использованием эмбедингов.
- Метод приближенного поиска ближайших соседей (ANN) – это метод ускорения плотного векторного поиска путем фильтрации документов, содержащих похожие векторы, перед выполнением дорогостоящих вычислений сходства между запросом и векторами каждого документа.
- Поиск по плотным векторам может быть значительно оптимизирован для скорости поиска, использования памяти и вызова лучших результатов путем объединения таких методов, как поиск ANN, квантизация, эмбединги Matryoshka Representation Learning (MRL) и избыточный запрос и реранкинг результатов.
- Би-кодировщики генерируют отдельные эмбединги для запросов и документов и поддерживают сопоставление и ранжирование большого объема, тогда как кросс-кодировщики требуют гораздо больше вычислений во время запроса и, таким образом, лучше всего используются для реранкинга меньшего количества лучших результатов, полученных из би-кодировщика или лексического поиска.

14

Ответы на вопросы с помощью тонко настроенной большой языковой модели

В этой главе рассматривается:

- создание приложения для ответов на вопросы с использованием LLM;
- создание набора данных вопрос-ответ для обучения;
- тонкая настройка LLM на основе трансформеров;
- интеграция конвейера NLP на основе глубокого обучения для извлечения и ранжирования ответов результата поиска.

Мы рассмотрели основы семантического поиска с использованием трансформеров в главе 13, поэтому теперь мы готовы попытаться решить одну из самых сложных проблем в поиске: ответы на вопросы.

Ответы на вопросы – это процесс возврата ответа на запрос пользователя, а не просто возврат списка результатов поиска, перечня документов. Существует два типа подходов к ответам на вопросы: экстрактивный и абстрактивный. *Экстрактивный метод ответов на вопросы* – это

процесс поиска точных ответов на вопросы из полученных документов (извлечение). Он возвращает фрагменты документов, содержащие вероятный ответ на вопрос пользователя, чтобы не нужно было проецировать результаты поиска. Напротив, *абстрактный метод ответов на вопросы* – это процесс генерации ответов на вопросы пользователя либо в виде резюме нескольких документов, либо напрямую из LLM без исходных документов. В этой главе мы сосредоточимся в первую очередь на извлечении ответов на вопросы, оставив сгенерированные ответы на вопросы для главы 15.

Решая проблему ответов на вопросы, вы достигнете трех вещей:

- вы лучше поймете инструментарий и экосистему трансформеров, с которыми начали знакомиться в главе 13;
- узнаете, как точно настроить большую языковую модель для конкретной задачи;
- объедините свою поисковую систему с передовыми методами естественного языка.

В этой главе мы покажем вам, как давать прямые ответы на вопросы и создать работающее приложение ответов на вопросы. Типы запросов, которые мы рассмотрим, – это вопросы с одним предложением – *кто, что, когда, где, почему и как* (who, what, when, where, why и how). Мы также продолжим использовать набор данных Stack Exchange outdoors из предыдущей главы. Наша цель – дать пользователям возможность задать ранее невиданный вопрос и получить краткий ответ, избавив пользователей от необходимости читать несколько результатов поиска, чтобы найти ответ самостоятельно.

14.1. Обзор модели вопрос–ответ

Традиционный поиск возвращает списки документов или страниц в ответ на запрос, но люди часто могут искать быстрый ответ на свой вопрос. В этом случае мы хотим избавить людей от необходимости копаться в блоках текста, когда в нашем контенте есть простой ответ.

В этом разделе мы представим задачу вопрос–ответ, а затем определим шаблон ретривер–ридер для реализации модели вопрос–ответ.

14.1.1. Как работает модель вопрос–ответ

Давайте рассмотрим, как модель вопрос–ответ работает на практике. В частности, мы реализуем *экстрактный метод ответов на вопросы*, который находит лучший ответ на вопрос в заданном тексте. Например, возьмем следующий вопрос:

Q: What are minimalist shoes?

Метод извлечения ответа на вопрос работает, просматривая большой документ, который, вероятно, содержит ответ, и определяет ответ для вас.

Давайте рассмотрим документ, который может содержать ответ на наш вопрос. Мы предоставляем вопрос `What are minimalist shoes?` и следующий текст документа (контекст) в модель:

There was actually a project done on the definition of what a minimalist shoe is and the result was "Footwear providing minimal interference with the natural movement of the foot due to its high flexibility, low heel to toe drop, weight and stack height, and the absence of motion control and stability devices". If you are looking for a simpler definition, this is what Wikipedia says, Minimalist shoes are shoes intended to closely approximate barefoot running conditions. 1 They have reduced cushioning, thin soles, and are of lighter weight than other running shoes, allowing for more sensory contact for the foot on the ground while simultaneously providing the feet with some protection from ground hazards and conditions (such as pebbles and dirt). One example of minimalistic shoes would be the Vibram FiveFingers shoes which look like this.

Документ можно разбить на множество небольших частей, известных как *интервалы* (spans), и модель извлекает лучший интервал в качестве ответа. Рабочая модель вопрос-ответ оценит вопрос и контекст и может выдать этот интервал в качестве ответа:

A: shoes intended to closely approximate barefoot running conditions

Но как модель узнает вероятность того, является ли какой-либо заданный промежуток ответом? Мы могли бы попытаться рассмотреть различные промежутки и посмотреть, все ли они каким-то образом представляют ответ, но это было бы очень сложно. Вместо этого задачу можно упростить, сначала узнав вероятность того, является ли каждый токен в контексте началом ответа, а также вероятность того, является ли каждый токен в контексте концом ответа. Поскольку мы смотрим только на вероятность того, что один токен представляет начало, а другой – конец, задачу легче понять и решить. Наши токены рассматриваются как дискретные значения, и экстрактивная (извлекающая) модель вопрос-ответ обучается изучать *функцию вероятности* (PMF), которая является функцией, дающей вероятность того, что дискретная случайная величина точно равна некоторому значению. Это отличается от измерения значений, которые являются непрерывными и используются в распределениях вероятностей, как мы обсуждали с непрерывным бета-распределением в главе 11. Основное различие между ними заключается в том, что наши токены являются дискретными значениями.

Используя эту стратегию, мы можем обучить одну модель, которая выучит две вероятностные функции массы – одну для начального токена диапазона ответа и одну для конечного токена диапазона ответа. Вы могли заметить, что мы сказали, что модель «может выдавать» предыдущий ответ. Поскольку модели, обученные с разными данными и гиперпараметрами, будут давать разные результаты, конкретный ответ, предоставленный для вопроса, может различаться в зависимости от параметров обучения модели.

Чтобы проиллюстрировать, как это работает, мы начнем с модели, которую кто-то уже обучил для извлекающей задачи ответа на вопрос. Модель выведет вероятность того, является ли токен началом или концом диапазона ответа. Когда мы определяем наиболее вероятное начало и конец ответа, это и есть наш диапазон ответа. Предварительно обученная модель, которую мы будем использовать, – это *deepset/goberta-base-squad2*, доступная в организации Hugging Face и обученная командой Deepset. Мы пропустим вопрос и контекст через эту модель и конвейер в листингах 14.1–14.3, чтобы определить начальные и конечные вероятности диапазона ответов, а также окончательный ответ. Рисунок 14.1 демонстрирует, как работает этот процесс путем *токенизации* входных данных вопроса и контекста, *кодирования* этих входных данных и *прогнозирования* наиболее подходящего диапазона ответов.

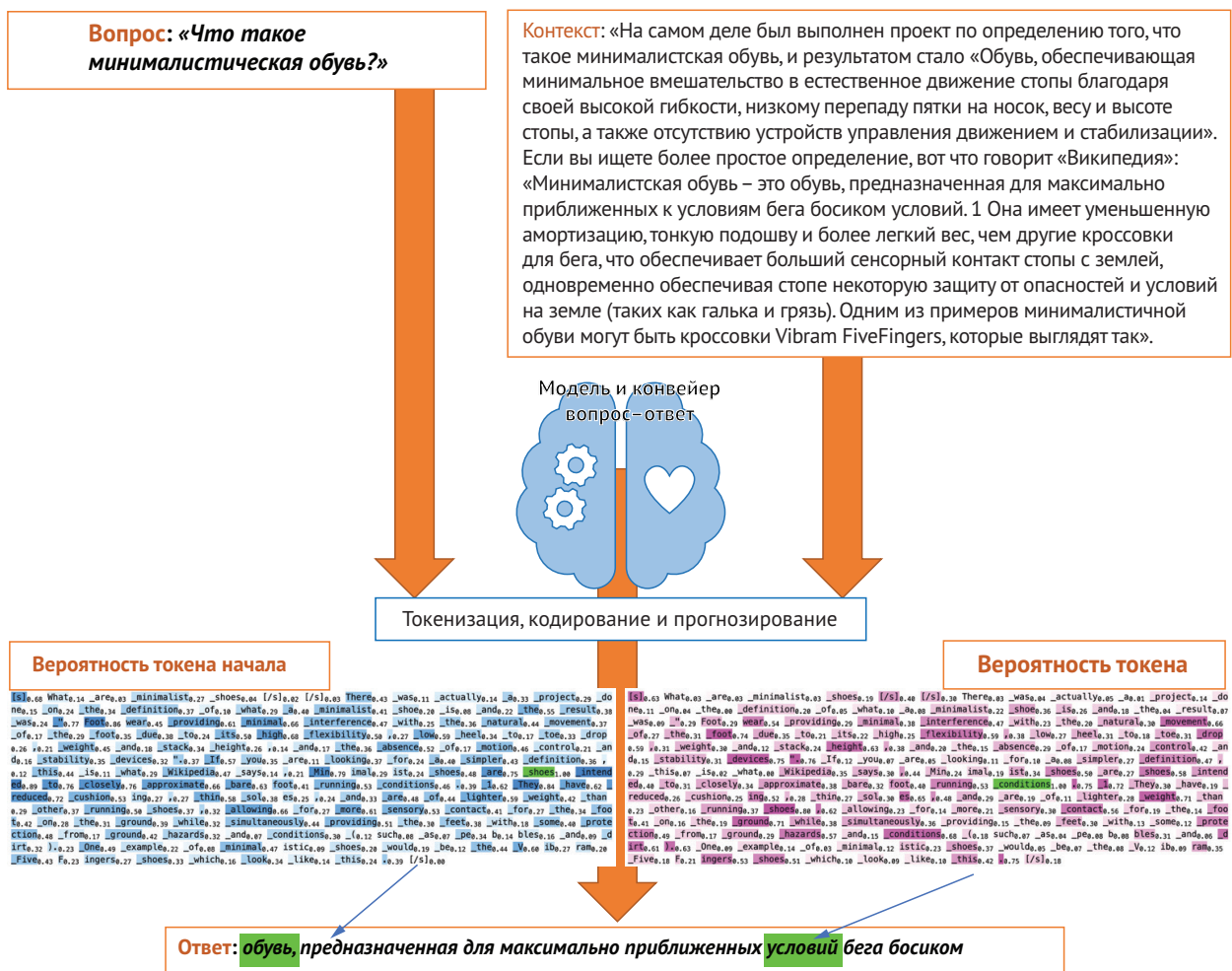


Рис. 14.1. Извлекающий процесс прогнозирование вопрос–ответ

На этом рисунке вы можете видеть, что вопрос и контекст сначала объединяются в пару для токенизации. Затем выполняется токенизация пары, получая входные данные токенов для модели. Модель принимает эти входные данные и затем выводит две последовательности: первая последовательность – это вероятности того, является ли каждый токен в контексте началом ответа, а вторая последовательность – это вероятности того, является ли каждый токен в контексте концом ответа. За-

тем последовательности начальных и конечных вероятностей объединяются для получения наиболее вероятного диапазона ответов.

В следующем листинге показан первый шаг этого процесса: токенизация.

Листинг 14.1. Загрузка токенизатора и модели

```
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
model_name = "deepset/roberta-base-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForQuestionAnswering.from_pretrained(model_name)
```

Имя модели в листинге 14.1 – это публичная модель, специально подготовленная для извлекаемых ответов на вопросы. Когда модель и токенизатор готовы, мы теперь можем передать пару вопрос–ответ, показанную в следующем листинге. Ответ покажет, что количество токенов равно количеству начальных и конечных вероятностей.

Листинг 14.2. Токенизация вопроса и контекста

```
question = "What are minimalist shoes"
context = """"There was actually a project done on the definition of what a
minimalist shoe is and the result was "Footwear providing minimal
interference with the natural movement of the foot due to its high
flexibility, low heel to toe drop, weight and stack height, and the absence
of motion control and stability devices". If you are looking for a simpler
definition, this is what Wikipedia says, Minimalist shoes are shoes intended
to closely approximate barefoot running conditions. 1 They have reduced
cushioning, thin soles, and are of lighter weight than other running shoes,
allowing for more sensory contact for the foot on the ground while
simultaneously providing the feet with some protection from ground
hazards and conditions (such as pebbles and dirt).
One example of minimalistic shoes would be the Vibram FiveFingers
shoes which look like this."""""

inputs = tokenizer(question, context, add_special_tokens=True,
                    return_tensors="pt")
input_ids = inputs["input_ids"].tolist()[0]

outputs = model(**inputs)
start_logits_norm = normalize(outputs[0].detach().numpy())
end_logits_norm = normalize(outputs[1].detach().numpy())

print(f"Total number of tokens: {len(input_ids)}")
print(f"Total number of start probabilities: {start_logits_norm.shape[1]}")
print(f"Total number of end probabilities: {end_logits_norm.shape[1]}")
```

Ответ:

```
Total number of tokens: 172
Total number of start probabilities: 172
Total number of end probabilities: 172
```


Входные данные получаются путем токенизации вопроса и контекста вместе. Выходные данные получаются путем выполнения прямого прохода с входными данными через модель. Переменная выходных данных – это список из двух предметов. Первый предмет содержит начальные вероятности, а второй предмет содержит конечные вероятности.

Рисунок 14.2 наглядно демонстрирует вероятности того, является ли каждый token в контексте вероятным началом диапазона ответа, а рис. 14.3 аналогичным образом демонстрирует, является ли каждый token вероятным концом диапазона ответа (более темное выделение указывает на более высокую вероятность).

[s] 0.68 What 0.14 _are 0.03 _minimalist 0.27 _shoes 0.04 [/s] 0.02 [/s] 0.03 There 0.43 _was 0.11 _actually 0.14 _a 0.33 _project 0.29 _do ne 0.15 _on 0.24 _the 0.34 _definition 0.37 _of 0.10 _what 0.29 _a 0.40 _minimalist 0.41 _shoe 0.20 _is 0.08 _and 0.22 _the 0.55 _result 0.38 _was 0.24 " 0.77 Foot 0.86 wear 0.45 _providing 0.61 _minimal 0.66 _interference 0.47 _with 0.25 _the 0.36 _natural 0.44 _movement 0.37 _of 0.17 _the 0.29 _foot 0.35 _due 0.38 _to 0.24 _its 0.50 _high 0.68 _flexibility 0.50 , 0.27 _low 0.59 _heel 0.34 _to 0.17 _toe 0.33 _drop 0.26 , 0.21 _weight 0.45 _and 0.18 _stack 0.34 _height 0.26 , 0.14 _and 0.17 _the 0.36 _absence 0.52 _of 0.17 _motion 0.46 _control 0.21 _an d 0.16 _stability 0.35 _devices 0.32 ". 0.37 If 0.57 _you 0.35 _are 0.11 _looking 0.37 _for 0.24 _a 0.40 _simpler 0.43 _definition 0.36 , 0.12 _this 0.44 _is 0.11 _what 0.29 _Wikipedia 0.47 _says 0.14 , 0.21 _Min 0.79 _imal 0.29 _ist 0.24 _shoes 0.48 _are 0.75 _shoes 1.00 _intend ed 0.89 _to 0.76 _closely 0.76 _approximate 0.66 _bare 0.63 _foot 0.41 _running 0.53 _conditions 0.46 , 0.39 _I 0.62 _They 0.84 _have 0.62 _ reduced 0.72 _cushion 0.53 _ing 0.27 , 0.27 _thin 0.58 _sol 0.38 _es 0.25 , 0.24 _and 0.33 _are 0.48 _of 0.44 _lighter 0.59 _weight 0.42 _than 0.29 _other 0.37 _running 0.50 _shoes 0.37 , 0.32 _allowing 0.66 _for 0.27 _more 0.61 _sensory 0.53 _contact 0.41 _for 0.27 _the 0.34 _foo t 0.42 _on 0.28 _the 0.31 _ground 0.39 _while 0.32 _simultaneously 0.44 _providing 0.51 _the 0.30 _feet 0.38 _with 0.18 _some 0.40 _prote ction 0.48 _from 0.17 _ground 0.42 _hazards 0.32 _and 0.07 _conditions 0.30 _ (0.12 _such 0.08 _as 0.07 _pe 0.34 _b 0.14 _bles 0.16 _and 0.09 _d irt 0.32) . 0.23 _One 0.49 _example 0.22 _of 0.08 _minimal 0.47 _istic 0.09 _shoes 0.20 _would 0.19 _be 0.12 _the 0.44 _V 0.60 _ib 0.27 _ram 0.20 _Five 0.43 _F 0.23 _ingers 0.27 _shoes 0.33 _which 0.16 _look 0.34 _like 0.14 _this 0.24 . 0.39 [/s] 0.00

Рис. 14.2. Вероятности того, является ли token началом диапазона ответа

[s] 0.63 What 0.03 _are 0.03 _minimalist 0.03 _shoes 0.19 [/s] 0.40 [/s] 0.30 There 0.03 _was 0.04 _actually 0.05 _a 0.01 _project 0.14 _do ne 0.11 _on 0.04 _the 0.00 _definition 0.20 _of 0.05 _what 0.10 _a 0.08 _minimalist 0.22 _shoe 0.36 _is 0.26 _and 0.18 _the 0.04 _result 0.07 _was 0.09 " 0.29 Foot 0.29 wear 0.54 _providing 0.29 _minimal 0.38 _interference 0.47 _with 0.23 _the 0.20 _natural 0.30 _movement 0.66 _of 0.27 _the 0.31 _foot 0.74 _due 0.35 _to 0.21 _its 0.22 _high 0.25 _flexibility 0.59 , 0.38 _low 0.27 _heel 0.31 _to 0.18 _toe 0.31 _drop 0.59 , 0.31 _weight 0.30 _and 0.12 _stack 0.24 _height 0.63 , 0.38 _and 0.20 _the 0.15 _absence 0.29 _of 0.17 _motion 0.24 _control 0.42 _an d 0.15 _stability 0.31 _devices 0.75 " . 0.76 If 0.12 _you 0.07 _are 0.05 _looking 0.11 _for 0.10 _a 0.08 _simpler 0.27 _definition 0.47 , 0.29 _this 0.07 _is 0.02 _what 0.00 _Wikipedia 0.35 _says 0.30 , 0.44 _Min 0.24 _imal 0.19 _ist 0.34 _shoes 0.50 _are 0.27 _shoes 0.58 _intend ed 0.40 _to 0.31 _closely 0.34 _approximate 0.38 _bare 0.32 _foot 0.40 _running 0.53 _conditions 1.00 , 0.75 _I 0.72 _They 0.30 _have 0.19 _ reduced 0.26 _cushion 0.25 _ing 0.52 _thin 0.27 _sol 0.30 _es 0.65 , 0.48 _and 0.29 _are 0.19 _of 0.11 _lighter 0.28 _weight 0.71 _than 0.23 _other 0.16 _running 0.37 _shoes 0.80 , 0.62 _allowing 0.23 _for 0.14 _more 0.21 _sensory 0.30 _contact 0.56 _for 0.19 _the 0.14 _foo t 0.41 _on 0.16 _the 0.19 _ground 0.71 _while 0.38 _simultaneously 0.36 _providing 0.15 _the 0.09 _feet 0.30 _with 0.13 _some 0.12 _prote ction 0.49 _from 0.17 _ground 0.29 _hazards 0.57 _and 0.15 _conditions 0.68 _ (0.18 _such 0.07 _as 0.04 _pe 0.08 _b 0.08 _bles 0.31 _and 0.06 _d irt 0.61) . 0.63 _One 0.09 _example 0.14 _of 0.03 _minimal 0.12 _istic 0.23 _shoes 0.37 _would 0.05 _be 0.07 _the 0.08 _V 0.12 _ib 0.09 _ram 0.35 _Five 0.18 _F 0.21 _ingers 0.53 _shoes 0.51 _which 0.10 _look 0.09 _like 0.10 _this 0.42 . 0.75 [/s] 0.18

Рис. 14.3. Вероятности того, является ли token концом диапазона ответа

Обратите внимание, что каждый token имеет начальную вероятность (на рис. 14.2) и конечную вероятность (на рис. 14.3) в соответствующем индексе. Мы также нормировали начальную и конечную вероятности в каждом индексе, чтобы они были в диапазоне от 0.0 до 1.0, что упрощает их понимание и вычисление. Мы называем эти списки начальных и конечных вероятностей *логитами*¹, поскольку они являются списками статистических вероятностей.

Например, для 17-го токена (_definition) вероятность того, что этот token будет началом ответа, составляет ~0.37, а вероятность того, что он будет концом ответа, составляет ~0.20. Поскольку мы нормировали оба

¹ Логиты в программировании, англ. *logit*, – это необработанные ненормированные прогнозы, генерируемые последним уровнем нейронной сети перед применением функции активации, часто используемой в задачах классификации. – *Прим. ред.*

списка, начало нашего диапазона ответов – это токен, где `start_logits_norm == 1.0`, а конец диапазона ответов – это токен, где `end_logits_norm == 1.0`.

В следующем листинге показано, как генерировать списки токенов на рис. 14.2 и 14.3, а также как извлечь окончательный диапазон ответов.

Листинг 14.3. Определение диапазона ответов из токенизированного контекста

```
start_tokens = []
end_tokens = []
terms = tokenizer.convert_ids_to_tokens(input_ids)
start_token_id = 0
end_token_id = len(terms)
for i, term in enumerate(terms):
    start_tokens.append(stylize(term, [0, 127, 255], start_logits_norm[0][i]))
    end_tokens.append(stylize(term, [255, 0, 255], end_logits_norm[0][i]))
    if start_logits_norm[0][i] == 1.0:
        start_token_id = i
    if end_logits_norm[0][i] == 1.0:
        end_token_id = i + 1

answer = terms[start_token_id:end_token_id]
display(HTML(f'<h3>{clean_token(" ".join(answer))}</h3>'))
display(HTML(f'<pre>{" ".join(start_tokens)}</pre>'))
display(HTML(f'<pre>{" ".join(end_tokens)}</pre>'))
```

Извлечение диапазона
ответов, показанное
в следующем выводе.

Конечные вероятности,
показанные на рис. 14.3.

Начальные вероятности,
показанные на рис. 14.2.

Вывод:

```
_shoes _intended _to _closely _approximate _bare foot _running _conditions
```

Подгоняя начальные и конечные массовые функции вероятности во время обучения к набору данных из вопросов, контекстов и триплетов ответов, мы создаем модель, которая может предоставить вероятности для наиболее вероятных ответов на новые вопросы и контексты. Затем мы используем эту модель в листинге 14.3 для выполнения вероятностного поиска, чтобы определить наиболее вероятный диапазон в тексте, который отвечает на вопрос.

На практике это работает следующим образом.

- 1 Мы выбираем минимальный и максимальный размер диапазона – где диапазон представляет собой набор непрерывных слов. Например, ответ может быть длиной в одно слово или, как в предыдущем ответе, он может быть длиной в восемь слов. Нам нужно установить эти размеры диапазона заранее.
- 2 Для каждого диапазона мы проверяем вероятность того, является ли диапазон правильным ответом. Ответом является диапазон с самой высокой вероятностью.

- 3 Когда мы закончим проверку всех диапазонов, мы представляем правильный ответ.

Построение модели требует множества триплетов вопрос–контекст–ответ и способа предоставления этих триплетов модели, чтобы можно было выполнять вычисления. Введите трансформер-кодировщик, с которым вы уже должны быть знакомы из главы 13. Сначала мы кодируем множество обучающих данных с помощью LLM, который создает плотные векторы. Затем обучаем нейронную сеть, чтобы узнать массовую функцию вероятности того, отвечает ли кодировка заданного диапазона на вопрос, используя положительные и отрицательные обучающие примеры.

Позже в этой главе мы увидим, как настроить модель вопрос–ответ, но сначала нам нужно рассмотреть очень важную деталь: когда кто-то задает вопрос, откуда мы получаем контекст?

14.1.2. Шаблон ретривер–ридер

Читая о том, как работает извлечение ответа на вопрос, вы могли подумать: «Итак, надо ли мне для каждого запроса проверить вероятности каждого диапазона во всем корпусе?» Нет! Это было бы чрезвычайно медленно и неправильно, поскольку у нас уже есть действительно быстрый и точный способ получения соответствующих документов, которые, вероятно, содержат ответ: поиск.

На самом деле мы собираемся сделать очень мощный текстовый маркер. Представьте всю систему вопрос–ответ как своего рода автоматического справочного библиотекаря. Он знает, какой документ содержит ваш ответ, а затем читает текст этого документа, чтобы указать вам точный ответ.

Это известно как шаблон ретривер–ридер¹. Этот шаблон использует один компонент для извлечения и ранжирования документов-кандидатов (выполнение запроса к поисковой системе) и другой компонент для чтения диапазонов наиболее релевантных документов и извлечения соответствующего ответа. Это очень похоже на то, как работает выделение в поисковых системах на основе Lucene, таких как Solr, OpenSearch или Elasticsearch: унифицированный маркер находит лучшие отрывки, содержащие проанализированные термины запроса, и использует их в качестве контекста. Затем он определяет точную локацию запрошенных ключевых слов в этом контексте, чтобы показать конечному пользователю окружающий контекст.

Мы собираемся создать что-то похожее на маркер, но вместо того, чтобы показывать контекст, содержащий запрошенные пользователем ключевые слова, наш вопросно-ответный маркер будет отвечать на такие вопросы:

¹ Ретривер в программировании – это часть системы, которая отвечает за поиск и извлечение релевантной информации. Он помогает приложениям эффективно получать доступ к необходимой информации из больших наборов данных без необходимости хранить сами документы. Ридер, англ. *Reader*, – это абстрактный класс, представляющий входной поток символов. Классы, наследующие *Reader*, позволяют читать текстовые данные из различных источников, таких как файлы или сетевые соединения. – *Прим. ред.*

Q: What are minimalist shoes?

A: shoes intended to closely approximate barefoot running conditions

Давайте рассмотрим полный контекст. Когда мы спрашиваем «Что такое минималистская обувь?», мы сначала используем ретривер, чтобы получить документ, который с наибольшей вероятностью будет содержать ответ. В этом случае был возвращен этот документ (здесь сокращенный, но полностью показанный в разделе 14.1.1):

There was actually a project done on the definition... this is what Wikipedia says, Minimalist shoes are shoes intended to closely approximate barefoot running conditions. 1 They have reduced cushioning, thin soles, ...

(На самом деле был выполнен проект по определению... вот что говорит «Википедия»: «Минималистская обувь – это обувь, предназначенная для максимально приближенных к условиям бега босиком условий. 1 У них уменьшенная амортизация, тонкая подошва...»)

Затем ридер получает документ, просматривает его и находит текст, который, скорее всего, содержит ответ на вопрос.

Помимо использования модных трансформеров для поиска правильного ответа, мы делаем шаг за пределы базового поиска, поскольку мы фактически используем уверенность ридера в вопросе–ответе в качестве ранкера. Поэтому, если мы не уверены, какой документ, скорее всего, содержит ответ на этапе извлечения, мы заставим ридер просмотреть кучу документов и посмотреть, какой из них является лучшим ответом. «Куча документов» будет нашим окном ранкинга, которое мы можем установить любого размера.

Однако помните, что анализ документов в реальном времени неэффективен. Мы не должны просить ридер просмотреть 100 документов – это займет слишком много времени. Мы ограничим его гораздо меньшим числом, например 3 или 5. Ограничение окна ридера заставляет нас убедиться, что наша поисковая система очень точна. Результаты должны быть релевантными, так как ретривер, который не получает релевантных кандидатов в размере окна из 5 лучших, не даст читателю ничего полезного для работы.

У ретривера–ридера есть две отдельные задачи, поэтому мы даже можем заменить наш ретривер чем-то другим. Мы показали, как можно использовать лексическую поисковую систему с готовым ранжированием (BM25, как описано в главе 3), но вы также можете попробовать это с плотным векторным индексом эмбедингов, как описано в главе 13.

Прежде чем мы сможем принимать вопросы живых пользователей, нам также нужно будет обучить модель вопрос–ответ для прогнозирования наилучшего ответа из контекста. Полный набор шагов, которые мы пройдем для создания нашего приложения вопрос–ответ, выглядит следующим образом.

- 1 *Настройте ретривер с помощью нашего поискового движка* – мы будем использовать простой запрос с высокой степенью отклика для возможных ответов для нашего примера.
- 2 *Соответствующим образом обработайте и маркируйте данные* – это включает в себя получение данных в правильном формате и получение первого прохода по нашим ответам из базовой предварительно обученной модели. Затем мы возьмем этот первый проход, вручную исправим его и отметим, какие примеры использовать для обучения и тестирования.
- 3 *Изучите нюансы структур данных* – мы будем использовать существующую структуру данных, которая будет представлять наши обучающие и тестовые данные в правильном формате для задачи тонкой настройки.
- 4 *Сделайте тонкую настройку модели* – используя исправления, которые мы сделали на предыдущем шаге, мы обучим тонко настроенную модель вопрос–ответ для большей точности, чем базовая.
- 5 *Используйте модель в качестве ридера во время запроса* – мы соберем все, что позволит нам запрашивать запрос, получать кандидатов из поисковой системы, считывать и переранжировать ответы из модели и отображать их в качестве ответа.

Рисунок 14.4. показывает поток всей архитектуры для нашего ретривера, ридера и ранкера.

- 1 Пользователь задает вопрос, и документы запрашиваются в ретривере (поисковике) с помощью вопроса.
- 2 Поисковая система сопоставляет и ранжирует, чтобы получить топ- k наиболее релевантных документов для ридера.
- 3 Исходный вопрос соединяется в пару с каждым топ- k извлеченным контекстом и отправляется в конвейер QA.
- 4 Пары вопрос–контекст токенизируются и кодируются в интервалы ридером, который затем прогнозирует топ- n наиболее вероятных интервалов спектра ответов с их вероятностями подобия в качестве оценки.
- 5 Ранкер сортирует баллы для каждого диапазона топ- n ответов в порядке убывания.
- 6 Самый высоко оцененный ответ от ранкера является принятым ответом и отображается пользователю.

Чтобы выполнить все это, нам нужно настроить ретривер, обработать данные для обучения модели ридера, настроить модель ридера с использованием этих данных и построить ранкер. Стратегии настройки ретривера (поисковой системы) уже были подробно рассмотрены в этой книге, поэтому на следующем этапе мы обработаем данные.

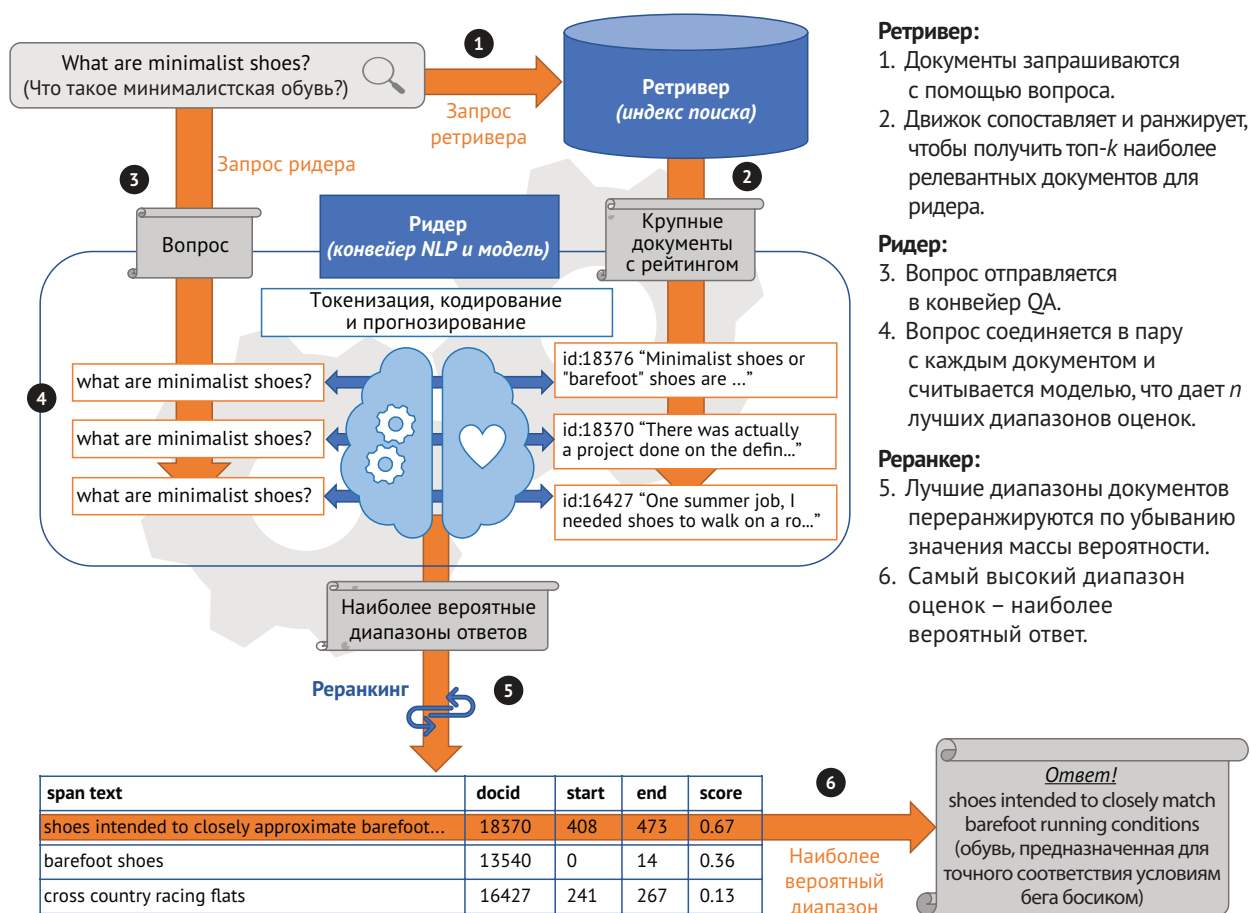


Рис. 14.4. Модель ретривера–ридера для извлечения ответа на вопрос

14.2. Создание обучающего набора данных для модели вопрос–ответ

В этом разделе мы создадим набор данных, который сможем использовать для обучения нашей модели вопрос–ответ. Это включает в себя несколько шагов.

- Сбор и очистка данных для соответствия проблемному пространству вопрос–ответ нашему контенту.
- Автоматическое создание набора silver (получищенного набора данных, требующего дальнейшей маркировки) из существующей модели и корпуса.
- Ручная корректировка набора silver для получения набора golden (надежного набора данных, который мы можем использовать для обучения).
- Разделение набора данных для обучения, тестирования и проверки точно настроенной модели.

Мы начнем с нашего набора данных Stack Exchange outdoors, поскольку его данные уже хорошо подходят для приложения вопрос–ответ. Нам нужны пары вопрос–ответ для использования при тонкой настройке базовой модели.

Набор данных *outdoors* уже хорошо отформатирован и состоит из небольших фрагментов вопросов и ответов. Благодаря мощи трансформеров мы можем взять готовые инструменты и модели и относительно быстро построить решение. Это гораздо проще, чем пытаться создать набор данных вопрос–ответ из чего-то другого, например книги *Great Expectations*. Если вы работаете с длинным текстом, например книгой или длинными документами, вам нужно сначала разбить текст на абзацы и вручную придумать вопросы для абзацев.

Наборы *golden* и *silver*

В машинном обучении набор *golden* – это точно маркированный набор данных, который используется для обучения, тестирования и проверки моделей. Мы относимся к наборам *golden* как к очень ценным активам, поскольку их сбор часто требует значительных ручных усилий. Точность и удобство использования обученной модели ограничены точностью и широтой набора *golden*. Таким образом, чем больше времени вы тратите на создание и проверку своего набора *golden*, тем лучше модель.

Чтобы сократить некоторые усилия, необходимые для маркировки данных, мы можем сэкономить время, позволив машине попытаться сгенерировать для нас маркированный набор данных. Этот автоматически сгенерированный набор маркированных данных называется набором *silver*, и он избавляет нас от необходимости начинать с нуля.

Набор *silver* не так надежен, как набор *golden*. Поскольку мы автоматически получаем набор *silver* через автоматизированные процессы, которые не так точны, как люди, будут ошибки. Таким образом, набор *silver* в идеале должен быть улучшен с помощью ручного аудита и исправлений для повышения его точности. Использование наборов *silver* для начальной загрузки вашего обучающего набора данных может сэкономить много времени и умственных усилий в долгосрочной перспективе, и это может помочь вам масштабировать курирование ваших обучающих данных.

14.2.1. Сбор и очистка набора данных вопрос–ответ

Перейдем к нашему первому шагу: давайте создадим набор данных, который мы можем маркировать и использовать для обучения модели. Для этого набора данных нам нужны вопросы со связанными контекстами, которые содержат их ответы. Листинг 14.4 показывает, как получить вопросы и контексты, которые содержат ответы, в строках кадра данных *pandas*. Нам нужно создать два запроса: один для получения вопросов сообщества и один для получения принятых ответов сообщества на эти вопросы. Мы будем использовать только пары вопрос–ответ, содержащие ответ, который принят. Мы выполним два запроса по отдельности и объединим их вместе. Модель, которую мы используем, ссылается на контент, из которого мы берем наш ответ в качестве контекста. Помните, мы не генерируем ответы, мы просто находим наиболее подходящий ответ внутри текста.

Листинг 14.4. Извлечение вопросов для обучения из Solr

```
def get_questions():
    question_types = ["who", "what", "when",
                     "where", "why", "how"]
    questions = []
    for type in question_types:
        request = {"query": type,
                  "query_fields": ["title"],
                  "return_fields": ["id", "url", "owner_user_id",
                                   "title", "accepted_answer_id"],
                  "filters": [("accepted_answer_id", "*")],
                  "limit": 10000}

        docs = outdoors_collection.search(**request)["docs"]
        questions += [document for document in docs
                      if document["title"].lower().startswith(type)]
    return questions
```

Сужает область типов вопросов, которые мы извлекаем.

Извлекает только вопросы, на которые есть принятые ответы.

Использует только заголовки, начинающиеся с типа вопроса.

Объединяет все ответы.

Со списком вопросов из листинга 14.4 нам далее нужно получить контексты, связанные с каждым вопросом. Листинг 14.5 возвращает датафрейм со следующими столбцами: `id`, `url`, `question` и `context`. Мы будем использовать `question` и `context` для генерации данных обучения и оценки для нашей модели вопрос-ответ в следующих разделах.

Листинг 14.5. Поиск принятых контекстов ответов

```
def get_answers_from_questions(questions, batch_size=500):
    answer_ids = list(set([str(q["accepted_answer_id"])
                          for q in questions]))
    batches = math.ceil(len(answer_ids) / batch_size)
    answers = {}
    for n in range(0, batches):
        ids = answer_ids[n * batch_size:(n + 1) * batch_size]
        request = {"query": "(" + " ".join(ids) + ")",
                  "query_fields": "id",
                  "limit": batch_size,
                  "filters": [("post_type", "answer")],
                  "order_by": [("score", "desc")]
        }
        docs = outdoors_collection.search(**request)["docs"]
        answers |= {int(d["id"]): d["body"] for d in docs}
    return answers

def get_context_dataframe(questions):
    answers = get_answers_from_questions(questions)
    contexts = {"id": [], "question": [], "context": [], "url": []}
    for question in questions:
        contexts["id"].append(question["id"])
        contexts["url"].append(question["url"])
        contexts["question"].append(question["title"]),
        contexts["context"].append(answers.get(question["accepted_answer_id"], ""))
```

Получает список всех отдельных идентификаторов ответов.

Рассчитывает количество поисковых запросов, которые необходимо сделать.

Извлекает все данные ответов для вопросов.


```

    if question["accepted_answer_id"] in answers:
        context = answers[question["accepted_answer_id"]]
    else:
        context = "Not found"
    contexts["context"].append(context)
return pandas.DataFrame(contexts)

questions = get_questions()
contexts = get_context_dataframe(questions)
display(contexts[0:5])

```

Вывод:

← Загружает во-
просы из ли-
стинга 14.4.

← Загружает контексты
для каждого вопроса.

id	question	context
4410	Who places the anchors that rock c...	There are two distinct styl...
5347	Who places the bolts on rock climb...	What you're talking about i...
20662	Who gets the bill if you activate ...	Almost always the victim ge...
11587	What sort of crane, and what sort ...	To answer the snake part of...
7623	What knot is this one? What are it...	Slip knot It's undoubtably ...

Мы рекомендуем вам изучить полный вывод для пар вопрос–контекст, чтобы оценить разнообразие входных данных и используемого языка. Мы также включили исходный URL, если вы хотите посетить веб-сайт Stack Exchange outdoors и самостоятельно изучить исходные данные в блокноте Jupyter.

14.2.2. Создание набора *silver*: автоматическая маркировка данных из предварительно обученной модели

Теперь, когда у нас есть наш набор данных, нужно его маркировать. Чтобы обучение работало, нам нужно сообщить модели, какой правильный ответ находится внутри контекста (документа), учитывая вопрос. Существует LLM, который уже неплохо справляется с выбором ответов: `deepset/goberta-base-squad2`. Эта модель была предварительно обучена компанией Deepset с использованием набора данных SQuAD2 и находится в свободном доступе на их странице Hugging Face (<https://huggingface.co/deepset>). SQuAD – это набор данных *Stanford Question Answering*, который представляет собой большой общедоступный набор данных, состоящий из тысяч пар вопросов и ответов. Команда Deepset начала с архитектуры RoBERTa (описанной в главе 13) и доработала модель на основе этого набора данных для задачи ответа на вопросы.

ПРИМЕЧАНИЕ. Неплохо ознакомиться с сайтом Hugging Face (<https://huggingface.co>). Сообщество Hugging Face очень активно и предоставило тысячи бесплатных предварительно обученных моделей, доступных для использования любым пользователем.

Наша стратегия заключается в использовании лучшей предварительно обученной модели, чтобы попытаться сначала ответить на все вопросы. Мы назовем эти ответы предположениями, а весь авто-

матически маркированный набор данных – набором silver. Затем мы пройдемся по предположениям набора silver и исправим их самостоятельно, чтобы получить набор golden.

В листинге 14.6 показана наша функция вопрос–ответ, которая использует конвейер трансформеров для вопросов и ответов и модель `deepset/roberta-base-squad2`. Мы используем их для построения конвейера с соответствующим токенизатором и целевым устройством (CPU или GPU). Это дает нам все необходимое для передачи необработанных данных и получения набора silver, как показано на рис. 14.5.

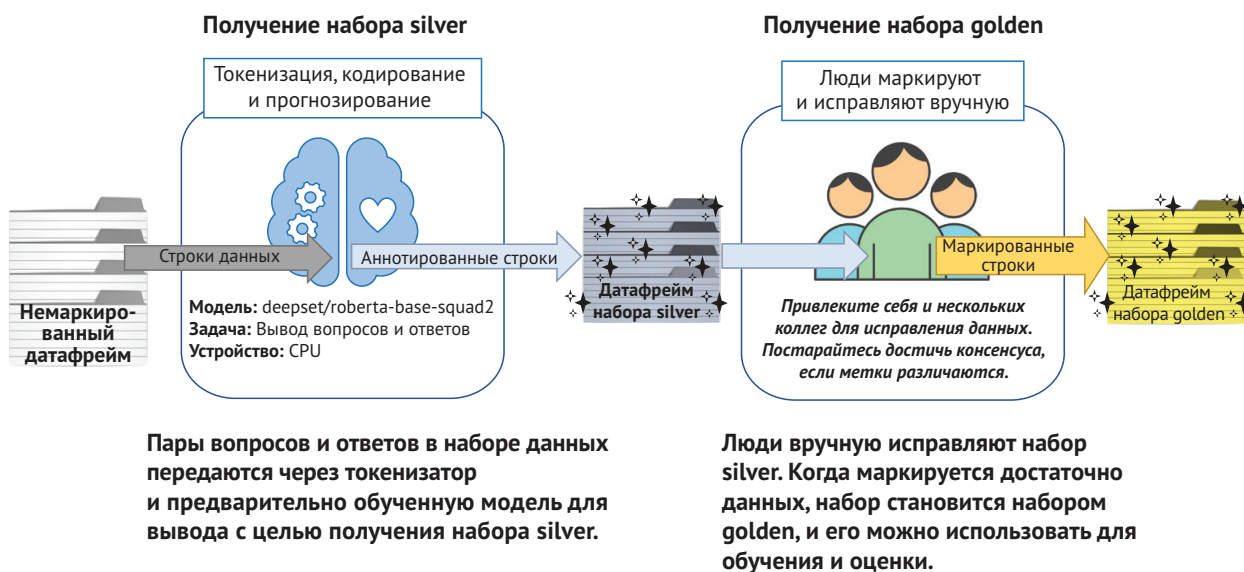


Рис. 14.5. Получение наборов silver и golden из подготовленного фрейма данных

В Python мы создаем функцию `answer_questions`, принимающую список контекстов, которые мы извлекли из нашего ретривера. Эта функция пропускает каждый вопрос и контекст через конвейер для генерации ответа и добавляет его в список. Мы не будем предполагать, что это на самом деле ответы на данном этапе, потому что многие из них будут неверными (как вы увидите, когда откроете файл). Мы будем считать что-то ответом только тогда, когда это будет проверено человеком. Такова природа обновления набора silver до набора golden.

Устройство (CPU или GPU) будет выбрано автоматически в зависимости от того, есть ли у вас GPU, доступный для вашей среды Docker, или нет. Сейчас самое время упомянуть, что, если вы запускаете или обучаете эти модели на домашнем компьютере с CPU или в конфигурации Docker, вам может потребоваться некоторое время для завершения вывода всех данных. Если вы не используете CPU, можете пропустить запуск листингов 14.6–14.7, так как мы уже предоставили выходные данные, необходимые для запуска последующих листингов в наборе данных этого блокнота. Листинг 14.6 генерирует набор silver для извлечения наиболее вероятных ответов для наших пар вопросов и принятых контекстов ответов, загруженных ранее в листинге 14.5.

Листинг 14.6. Генерация ответов по парам вопрос–контекст

```

from transformers import pipeline
import torch
import tqdm

def get_processor_device():
    return 0 if torch.cuda.is_available() else -1

def answer_questions(contexts, k=10):
    nlp = pipeline("question-answering", model=model_name,
                  tokenizer=model_name, device=device)

    guesses = []
    for _, row in tqdm.tqdm(contexts[0:k].iterrows(), total=k):
        result = nlp({"question": row["question"],
                     "context": row["context"]})
        guesses.append(result)
    return guesses

model_name = "deepset/roberta-base-squad2"
device = get_processor_device()

guesses = answer_questions(contexts, k=len(contexts))
display_guesses(guesses)

```

← Это конвейер, который мы проиллюстрировали на рис. 14.1.

← tqdm выводит ход выполнения операции в виде индикатора выполнения.

← Обработать с помощью GPU (CUDA), если доступно; в противном случае использовать CPU.

← Это конвейер, который мы проиллюстрировали на рис. 14.1.

← tqdm выводит ход выполнения операции в виде индикатора выполнения.

← Получает ответ (и оценку уверенности) для каждой пары вопрос–контекст.

← Обработать с помощью GPU (CUDA), если доступно; в противном случае использовать CPU.

Вывод:

```

score    start  end  answer
0.278927 474   516  a local enthusiast or group of enthusiasts
0.200848 81    117  the person who is creating the climb
0.018632 14    24   the victim
...
0.247008 227   265  the traditional longbow made from wood
0.480407 408   473  shoes intended to closely approximate barefoot
run...
0.563754 192   232  a tube of lightweight, stretchy material

```

Рекомендуется GPU

Вы можете свободно запускать эти листинги на своем персональном компьютере, но будьте осторожны – некоторые из них требуют много времени на CPU. Например, общее время выполнения наших тестов для листинга 14.6 сокращается примерно в 20 раз при запуске на GPU по сравнению с CPU среднего уровня. Обратите внимание, что примеры тонкой настройки далее в главе значительно выиграют от наличия GPU. Если у вас нет доступа к GPU для этих листингов, это нормально – мы обучили модель и уже включили ее для вас как часть набора данных на outdoors. Вы можете следить за листингами, чтобы увидеть, как обучается модель, и, если у вас нет GPU, можете просто пропустить

их запуск. Вы также можете использовать бесплатные сервисы, такие как Google Colab, или арендовать сервер с GPU у облачного провайдера, что обычно стоит несколько долларов США в час.

Если вам интересно узнать больше о графических процессорах и о том, почему они лучше подходят для таких задач, как обучение моделей, мы рекомендуем ознакомиться с книгой «Параллельные и высокопроизводительные вычисления» Роберта Роби и Юлианы Заморы (Manning, 2021).

14.2.3. Обучение с участием человека: ручная коррекция набора *silver* для получения набора *golden*

Файл CSV набора *silver* (`question-answering-squad2-guesses.csv`) используется в качестве первого шага при попытке ответить на вопросы. Мы будем использовать его с ручной коррекцией и маркировкой данных человеком для проработки набора *silver* до набора *golden*.

ПРИМЕЧАНИЕ. Ни один код Python не может сгенерировать набор *golden* для вас. Данные должны быть маркированы *человеком* (или, возможно, однажды моделью ИИ, высоко оптимизированной для принятия решений о релевантности) с пониманием предметной области. Все дальнейшие листинги будут использовать этот набор *golden*. Однако мы даем вам передышку, поскольку уже разметили данные для вас. Для справки: потребовалось от 4 до 6 часов, чтобы разметить около 200 предположений, полученных с помощью модели `deepset/roberta-base-squad2`.

Разметка данных самостоятельно даст вам более глубокое понимание сложности этой задачи NLP. Мы *настоятельно рекомендуем* вам маркировать как можно больше документов и повторно выполнить предстоящие задачи тонкой настройки. Оценка усилий, необходимых для получения качественных данных, и их влияние на точность модели – это урок, который вы можете усвоить только из собственного опыта.

Однако, прежде чем погрузиться в процесс и просто маркировать данные, нам нужно иметь план того, как и что маркировать. Для каждой строки нам нужно ее классифицировать и при необходимости самостоятельно записать правильный ответ в другой столбец.

Вот ключ, показанный на рис. 14.6, который мы использовали для всех размеченных строк в поле класса:

- -2 = это отрицательный пример (пример, в котором мы знаем, что предположение неверно!);
- -1 = проигнорировать этот вопрос, так как он слишком расплывчатый или нам не хватает какой-то информации. Например, *What is this bird?* (Что это за птица?). Мы не можем ответить на этот вопрос без изображения птицы, поэтому мы даже не пытаемся;
- 0 = это пример, который был исправлен человеком, чтобы выделить лучший диапазон ответов в том же контексте. Догадка, данная `deepset/roberta-base-squad2`, была неверной или неполной, поэтому мы ее изменили;

- 1 = это пример, на который deepset/roberta-base-squad2 дал правильный ответ, поэтому мы не изменили ответ;
- (пусто) = мы не проверяли эту строку, поэтому проигнорируем ее.

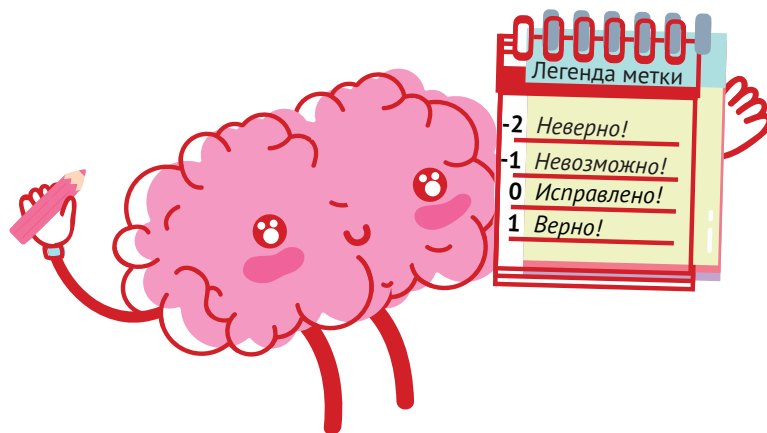


Рис. 14.6. Легенда¹ классов меток

Вам следует открыть файл `outdoors_golden_answers.csv` и самостоятельно просмотреть строки. Изучите соотношение вопросов, которые мы поместили как 0 и 1. Вы даже можете попробовать открыть файл в `pandas` и провести небольшой анализ, чтобы ознакомиться с золотым набором.

14.2.4. Форматирование набора *golden* для обучения, тестирования и проверки

Теперь, когда у нас есть маркированные данные, мы почти готовы обучить нашу модель, но сначала нужно перевести данные в правильный формат для конвейера обучения и оценки. Как только наши данные будут в правильном формате, нам также нужно будет разделить их на обучающие, тестовые и проверочные наборы для использования при обучении модели, чтобы убедиться, что она не переобучается нашим данным.

Преобразование маркированных данных в стандартизированный формат данных

Hugging Face предоставляет библиотеку под названием `datasets`, которую мы будем использовать для подготовки наших данных. Библиотека `datasets` может принимать имена многих общедоступных наборов данных и предоставлять стандартный интерфейс для работы с ними. Набор данных SQuAD2 является одним из доступных наборов данных, но, поскольку наш набор *golden* находится в пользовательском формате, сначала нужно преобразовать его в формат конфигурации стандартизированных наборов данных, показанный в следующем листинге.

¹ Легенда в программировании – это область графика, которая описывает каждую из частей графика и помогает лучше понять содержимое данных. Другими словами, «табличка». – Прим. ред.

Листинг 14.7. Преобразование набора данных golden в формат SQuAD

```

from datasets import Dataset, DatasetDict

def get_training_data(filename):
    golden_answers = pandas.read_csv(filename)
    golden_answers = golden_answers[golden_answers["class"] != None]
    qa_data = []
    for _, row in golden_answers.iterrows():
        answers = row["gold"].split("|")
        starts = [row["context"].find(a) for a in answers]
        missing = -1 in starts
        if not missing:
            row["title"] = row["question"]
            row["answers"] = {"text": answers, "answer_start": starts}
            qa_data.append(row)
    columns = ["id", "url", "title", "question", "context", "answers"]
    df = pandas.DataFrame(qa_data, columns=columns) \
        .sample(frac=1, random_state=0)
    train_split = int(len(df) * 0.75)
    eval_split = (int((len(df) - train_split) / 1.25) +
                  train_split - 1)
    train_dataset = Dataset.from_pandas(df[:train_split])
    test_dataset = Dataset.from_pandas(df[train_split:eval_split])
    validation_dataset = Dataset.from_pandas(df[eval_split:])
    return DatasetDict({"train": train_dataset,
                        "test": test_dataset,
                        "validation": validation_dataset})

datadict = get_training_data("data/outdoors/outdoors_golden_answers.
csv")
model_path = "data/question-answering/question-answering-training-set"
datadict.save_to_disk(model_path)

```

Случайно сортирует все примеры.

75 % примеров будут использоваться для обучения. Это даст нам 125 обучающих образцов.

20 % примеров будут использоваться для тестирования. Мы вычитаем 1 из train_split, чтобы разрешить 125/32/10 записей в трех разделах.

Оставшиеся 5 % примеров будут использоваться для проверки. Это будет 10 образцов.

SQuAD требует три группы данных: обучение, тестирование и проверка.

Первая часть функции в листинге 14.7 загружает CSV в датафрейм pandas и выполняет некоторую предварительную обработку и форматирование. После форматирования данные разделяются на три части и преобразуются.

Объект, возвращаемый и сохраняемый из листинга 14.7, представляет собой словарь набора данных (datadict), который содержит наши три раздела для обучения, тестирования и проверки. Для нашей таблицы данных с разделением, определенным в get_training_data, у нас есть 125 обучающих примеров, 32 тестовых примера и 10 проверочных примеров.

Избегание переобучения с тестовым набором и проверочным набором holdout

Переобучение модели (overfitting) означает, что вы обучили ее запоминать только предоставленные обучающие примеры. Это означает, что она не будет достаточно хорошо обобщать при обработке ранее не виданных данных¹.

Чтобы предотвратить переобучение, нам нужно было разделить наш набор данных на отдельные обучающие, тестовые и проверочные срезы, как мы сделали в листинге 14.7. Тестовые и контрольные проверочные наборы используются для измерения успешности модели после ее обучения. После того как вы пройдете весь процесс от начала до конца, рассмотрите возможность маркировки дополнительных данных и выполнения различных разделений по обучающим, тестовым и проверочным срезам, чтобы увидеть, как работает модель.

Мы используем разделение обучение–тест, чтобы предоставить некоторые данные для обучения модели, а некоторые – для проверки результата. Мы итеративно настраиваем гиперпараметры для обучения модели, чтобы достичь более высокой точности (измеренной с помощью функции потерь), когда модель применяется к тестовому набору.

Контрольный проверочный набор – это реальный прокси для невидимых данных, и он не проверяется до конца. После завершения обучения и тестирования вы затем проверяете окончательную версию модели, применяя ее к контрольным примерам. Если эта оценка намного ниже окончательной точности теста, ваша модель переобучена.

ПРИМЕЧАНИЕ. Количество используемых нами примеров довольно мало (125 обучающих примеров, 32 тестовых примера и 10 контрольных примеров проверки) по сравнению с тем, что вы бы использовали для тонкой настройки данных для системы, ориентированной на клиента. Как правило, ориентируйтесь на 500–2000 маркированных примеров. Иногда можно обойтись меньшим количеством, но обычно чем больше, тем лучше. Это потребует значительных временных затрат, но оно того стоит.

14.3. Тонкая настройка модели вопрос–ответ

Теперь мы рассмотрим получение лучшей модели путем тонкой настройки существующей модели `deepset/roberta-base-squad2` с помощью нашего набора `golden`.

К сожалению, этот следующий блокнот на CPU может работать довольно медленно. Если вы просматриваете листинги на машине с под-

¹ Переобучение происходит, когда модель обучается настолько хорошо на тренировочных данных, что запоминает неважные детали и «шумы» вместо того, чтобы обобщать закономерности. Это приводит к тому, что при работе с новыми данными модель работает плохо. – *Прим. ред.*

держкой CUDA и можете настроить среду Docker для использования графических процессоров, то у вас все должно быть готово! В противном случае мы рекомендуем вам использовать такой сервис, как Google Colab, который предлагает простой запуск блокнотов Jupyter на графических процессорах бесплатно, или другого поставщика облачных вычислений или хостинга, у которого есть готовое к работе устройство CUDA. Вы можете загрузить блокнот напрямую из Google Colab и запустить его без каких-либо других зависимостей, кроме нашего набора данных. Ссылка приведена выше листинга 14.8 в соответствующем блокноте.

СОВЕТ. Как мы уже отмечали ранее, если вы не хотите заниматься настройкой среды, совместимой с графическим процессором, вы также можете следовать листингам 14.8–14.13, не запуская их, поскольку мы уже обучили модель и включили ее для использования. Однако, если вы можете, мы настоятельно рекомендуем вам приложить усилия для получения доступа к GPU и самостоятельного обучения модели, чтобы увидеть, как работает этот процесс, и чтобы вы могли повозиться с гиперпараметрами. На рис. 4.7 показано ускорение, которое могут обеспечить GPU для массивно-параллельных вычислений, таких как обучение языковой модели.

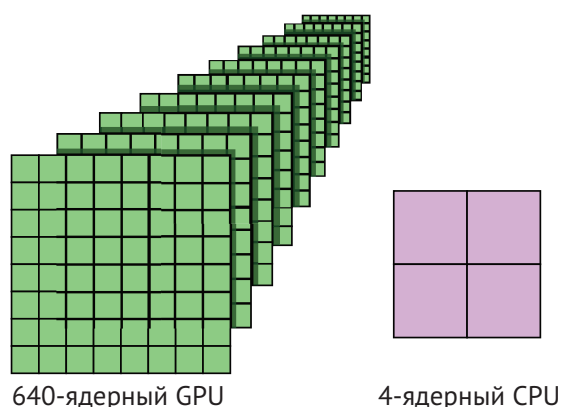


Рис. 14.7. V100 GPU (обычно доступен у облачных провайдеров) имеет 640 тензорных вычислительных ядер по сравнению с 4-ядерным x86-64 CPU. Tesla T4 имеет 2560 тензорных вычислительных ядер. По отдельности ядра CPU более мощные, но большинство GPU имеют на два-три порядка больше ядер, чем CPU. Это важно при выполнении массивно-параллельных вычислений для миллионов параметров модели

Первое, что нам нужно сделать, – это запросить доступ к устройству GPU. Код в следующем листинге инициализирует и возвращает идентификатор устройства доступного процессора. Если графический процессор настроен и доступен, мы должны увидеть идентификатор этого устройства. Если вы используете Colab и у вас возникли проблемы с листингом 14.8, может потребоваться изменить тип среды выполнения на GPU в настройках.

Листинг 14.8. Обнаружение и инициализация GPU

```
def get_processor_type():
    gpu_device = torch.device("cuda:0")
    cpu_device = torch.device("cpu")
    return gpu_device or cpu_device
```

```
def get_processor_device():
    return 0 if torch.cuda.is_available() else -1

print("Processor: " + str(get_processor_type()))
print("Device id: " + str(get_processor_device()))
```

Вывод:

```
Processor: device(type='cuda', index=0)
Device id: 0
```

У нас есть графический процессор (по крайней мере, в этом выводе листинга). В ответе `device(type='cuda', index=0)` – это то, что мы искали. Если графический процессор недоступен при запуске листинга, вместо него будет возвращен `device(type='cpu')`, что означает, что для обработки будет использоваться центральный процессор. Если у вас более одного доступного устройства для ноутбука, он перечислит каждое из них с увеличивающимся числовым идентификатором. Вы можете получить доступ к устройству позже в обучении, указав идентификатор (в нашем случае 0).

Когда наше устройство готово к работе, мы загрузим и токенизируем наш ранее маркированный набор данных из листинга 14.7.

14.3.1. Токенизация и формирование наших маркированных данных

Обучающий модуль модели не распознает слова; он распознает токены, которые существуют в словаре RoBERTa. Мы рассмотрели токенизацию в главе 13, где использовали ее в качестве начального шага при кодировании документов и запросов в плотные векторы для семантического поиска. Аналогично нам нужно токенизировать наш вопросно-ответный набор данных, прежде чем мы сможем использовать его для обучения модели. Модель принимает значения токенов в качестве входных параметров, как и любая другая модель трансформера.

В следующем листинге показано, как мы будем токенизировать данные перед обучением модели.

Листинг 14.9. Токенизация нашего обучающего набора

```
# This function adapted from:
# https://github.com/huggingface/notebooks/blob/master/examples/
# question_answering.ipynb
# Copyright 2001, Hugging Face. Apache 2.0 Licensed.
from datasets import load_from_disk
from transformers import RobertaTokenizerFast

file = "data/question-answering/question-answering-training-set"
datadict = datasets.load_from_disk(file)
tokenizer = from_pretrained("roberta-base")
...
```

Загружает datadict, который мы создали в листинге 14.7 из нашего набора golden.

Загружает предварительно обученный токенизатор (roberta-base).

Это будет количество токенов как в вопросе, так и в контексте.

Иногда нам нужно разбить контекст на более мелкие чанки, чтобы эти чанки перекрывались на этом количестве токенов.

Добавляет заполняющие токены в конец для пар вопрос-контекст, которые короче размера входных данных модели.

Выполняет токенизацию для каждого из примеров.

```
def tokenize_dataset(examples):
    maximum_tokens = 384
    document_overlap = 128
    pad_on_right = tokenizer.padding_side == "right"
    tokenized_examples = tokenizer(
        examples["question" if pad_on_right
                  else "context"],
        examples["context" if pad_on_right
                  else "question"],
        truncation="only_second" if pad_on_right
                               else "only_first",
        max_length=maximum_tokens,
        stride=document_overlap,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding="max_length"
    )
    ...
    return tokenized_examples
```

Дополнительная обработка для определения начальных и конечных позиций для вопросов и контекстов. Полный алгоритм см. в блокноте.

```
tokenized_datasets = datadict.map(
    tokenize_dataset,
    batched=True,
    remove_columns=datadict["train"].column_names)
```

Вызывает токенизатор для каждого примера в нашем наборе данных golden.

Мы загружаем токенизатор (обученный на основе модели roberta-base), загружаем наш question-answering-training-set набор golden с диска (data/question-answering/question-answering-training-set/), а затем запускаем примеры из набора golden через токенизатор, чтобы сгенерировать объект tokenized_datasets с обучающими и тестовыми наборами данных, которые мы вскоре передадим модели-тренеру¹.

Для каждого контекста мы генерируем список тензоров с определенным количеством эмбедингов на тензор и определенным количеством чисел с плавающей точкой на эмбединг. Форма тензоров, содержащих токены, должна быть одинаковой для всех примеров, которые мы предоставляем тренеру и оценщику. Мы достигаем этого с помощью техники скользящего окна.

Метод скользящего окна – это техника, которая включает в себя разбиение длинного списка токенов на множество подписков токенов, но где каждый подсписок после первого разделяет перекрывающееся количество токенов с предыдущим подписком. В нашем случае maximum_

¹ Модель-тренер – это инструмент для быстрого обучения и оценки различных моделей, а также использования обученных моделей для классификации данных. Это высокоуровневый API, который упрощает процесс обучения и точной настройки моделей машинного обучения. Он позволяет пользователям сосредоточиться на разработке и экспериментировании с моделями, а не управлять сложными деталями процесса обучения. – Прим. ред.

tokens определяет размер каждого подписка, а document_overlap определяет перекрытие. Этот процесс скольжения окон показан на рис. 14.8.

Рисунок 14.8 демонстрирует очень маленькие числа maximum_tokens (24) и document_overlap (8) для иллюстративных целей, но реальный процесс токенизации разбивает контексты на тензоры по 384 токена с перекрытием 128.

Техника скольжения окон также использует *заполнение* (padding), чтобы гарантировать, что каждый тензор имеет одинаковую длину. Если количество токенов в последнем тензоре контекста меньше максимального (384), то остальные позиции в тензоре заполняются пустым маркерным токеном, так что окончательный размер тензора (tensor size) также равен 384¹.

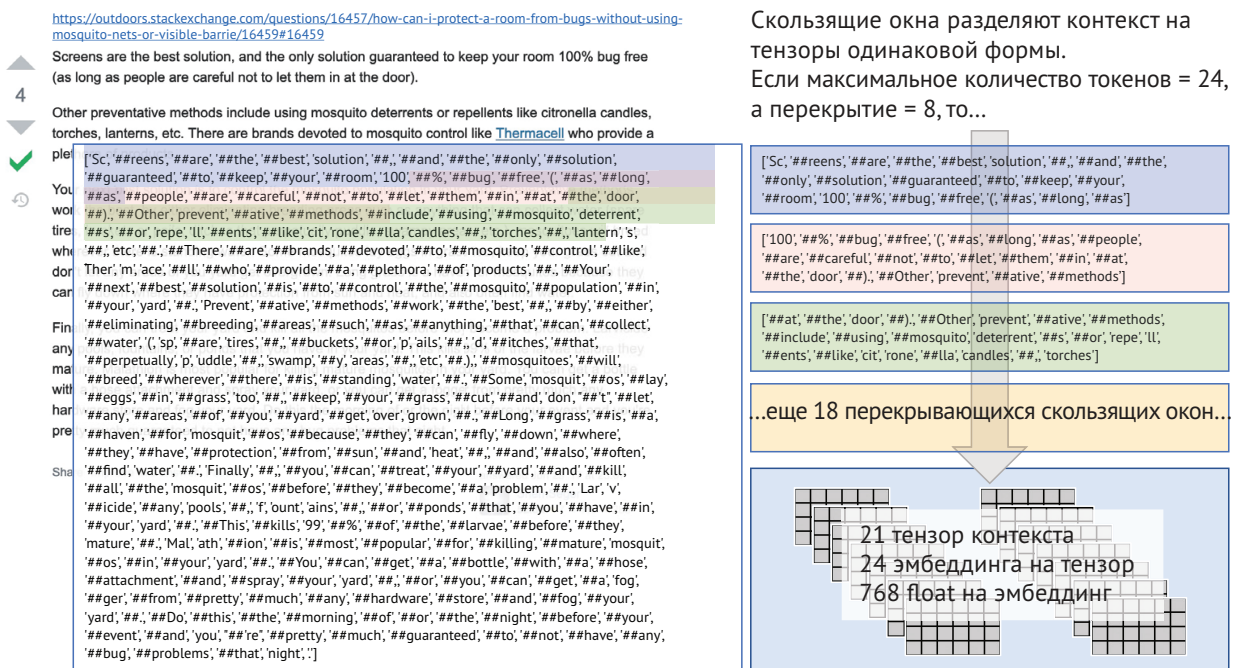


Рис. 14.8. Визуализация техники скользящего окна, которая разделяет один контекст на тензоры одинаковой формы

Важно знать, как обрабатываются контексты, поскольку это может повлиять как на точность, так и на время обработки. Если мы пытаемся идентифицировать ответы в длинных документах, процесс скольжения окон может снизить точность, особенно если maximum_tokens и document_overlap малы и, таким образом, слишком сильно фрагментируют контекст. Длинные документы также будут нарезаны на несколько тензоров, которые в совокупности потребуют больше времени для обработки. Большинство контекстов в наборе данных outdoors соответствуют указанным нами максимумам, но эти компромиссы важно учитывать в других наборах данных при выборе параметров maximum_tokens и document_overlap.

¹ Tensor size – это общее количество элементов в тензоре. Это произведение всех размеров тензора. Другими словами, он представляет общий объем данных, которые хранятся в тензоре. – Прим. ред.

14.3.2. Настройка модели-тренера

У нас есть последний шаг перед обучением нашей модели: нужно указать, как будет происходить обучение и оценка.

При обучении нашей модели нам нужно указать базовую модель и аргументы обучения (гиперпараметры), а также наши обучающие и тестовые наборы данных. Вам нужно будет понять следующие ключевые понятия при настройке гиперпараметров для модели-тренера:

- эпохи – сколько раз тренер будет делать итерации по набору данных. Больше эпох помогает усилить контекст и сократить потери с течением времени. Однако слишком большое количество эпох, скорее всего, приведет к переобучению вашей модели, и 3 эпохи – это распространенный выбор при тонкой настройке трансформеров;
- размеры пакета – количество примеров, которые будут обучены-оценены одновременно. Большой размер пакета может привести к получению лучшей модели. Этот параметр ограничен количеством ядер графического процессора и доступной памятью, но общепринятой практикой является размещение как можно большего количества данных в партии, чтобы максимально использовать доступные ресурсы;
- прогрев¹ – при обучении модели может быть полезно медленно настраивать модель изначально, чтобы ранние примеры не оказывали чрезмерного влияния на изученные параметры модели. Шаги прогрева позволяют постепенно улучшать модель (по параметру скорости обучения), что помогает предотвратить переобучение обучающего алгоритма на ранних примерах;
- затухание – затухание веса используется для уменьшения переобучения путем умножения каждого веса на это постоянное значение на каждом шаге. Обычно в качестве затухания веса используют 0.01, но его можно изменить на большее значение, если модель быстро переобучается, или на меньшее значение, если вы не видите достаточно быстрого проявления этого эффекта.

В листинге 4.10 показана настройка обучающего алгоритма модели. Гиперпараметры (`training_args`), указанные в листинге, используются SQuAD2 по умолчанию, но вы можете свободно изменять любой из них, чтобы увидеть, как это улучшает качество модели вопрос-ответ для ваших собственных вопросов.

При попытке выбрать наилучшие настройки распространенной методикой является выполнение поиска по сетке по этим гиперпараметрам. *Поиск по сетке* – это процесс, который автоматически перебирает значения параметров и проверяет, как настройка каждого из них в различных комбинациях улучшает качество обучен-

¹ Прогрев, англ. *warmup*, в программировании – это процесс подготовки приложения к запуску, который включает компиляцию и оптимизацию кода, в случае больших и сложных приложений может занимать до нескольких минут. – *Прим. ред.*

ных моделей. Мы включаем пример поиска по сетке в прилагаемые блокноты, если вы хотите глубже погрузиться в настройку параметров, но сейчас мы продолжим с гиперпараметрами, указанными в листинге 14.10.

Листинг 14.10. Инициализация тренера и его гиперпараметров

```
from transformers import RobertaForQuestionAnswering, TrainingArguments,
    Trainer, default_data_collator

model = RobertaForQuestionAnswering.from_pretrained(
    "deepset/roberta-base-squad2")

training_args = TrainingArguments(
    evaluation_strategy="epoch",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=64,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="data/question-answering/logs",
    output_dir="data/question-answering/results")

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=default_data_collator,
    tokenizer=tokenizer,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["test"])
```

Размер пакета для оценки. (указывает на `per_device_train_batch_size` и `per_device_eval_batch_size`)

Быстро-та снижения веса. (указывает на `warmup_steps`)

Оценивает потери на эпоху. (указывает на `evaluation_strategy="epoch"`)

Общее количество эпох обучения. (указывает на `num_train_epochs=3`)

Размер пакета на устройство во время обучения. (указывает на `per_device_train_batch_size=16` и `per_device_eval_batch_size=64`)

Количество шагов прогрева для планировщика скорости обучения. (указывает на `warmup_steps=500`)

Инстанцированная модель трансформеров Hugging Face для обучения. (указывает на `model=model`)

Аргументы обучения. (указывает на `args=training_args`)

Указывает тренировочный датасет. (указывает на `train_dataset=tokenized_datasets["train"]`)

Указывает набор данных оценки. (указывает на `eval_dataset=tokenized_datasets["test"]`)

14.3.3. Делаем обучение и оцениваем потери

После того как наши гиперпараметры настроены, пришло время обучить модель. Следующий листинг запускает ранее настроенный тренер, возвращает выходные данные обучения, показывающие производительность модели, и сохраняет модель.

Листинг 14.11. Обучение и сохранение модели

```
trainer.train()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
trainer.save_model(model_name)
```

Вывод:

```
[30/30 00:35, Epoch 3/3]
Epoch   Training Loss   Validation Loss   Runtime   Samples Per Second
1        No log         2.177553         1.008200   43.642000
2        No log         2.011696         1.027800   42.811000
3        No log         1.938573         1.047700   41.996000
```

```
TrainOutput(global_step=30, training_loss=2.531823984781901,
             metrics={'train_runtime': 37.1978,
                      'train_samples_per_second': 0.806,
                      'total_flos': 133766734473216, 'epoch': 3.0})
```

Функция *потерь* – это функция принятия решений, которая использует ошибку для получения количественной оценки того, насколько плоха модель. Более низкие потери означают более высокое качество модели. Мы ищем постепенное снижение потерь с каждой эпохой, что указывает на то, что модель продолжает становиться лучше с большим обучением. Мы перешли от проверочных потерь 2.178 к 2.012 и 1.939 на нашем тестовом наборе. Все числа уменьшаются с постоянной скоростью (без резких скачков), и это хороший знак.

Общая потеря обучения для этой недавно настроенной модели составляет 2.532, а потеря проверки на нашем тестовом наборе составляет 1.939. Учитывая ограничения нашего небольшого набора данных для тонкой настройки и конфигурации гиперпараметров, потеря проверки в размере 1.939 является довольно хорошей.

14.3.4. Валидация и подтверждение с удерживанием

Как узнать, можно ли успешно использовать нашу обученную модель для ответа на вопросы в реальном мире? Что ж, нам нужно протестировать модель на нашем наборе отложенных данных для проверки¹. Напомним, что набор отложенных данных для проверки – это третий набор данных (всего с 10 примерами) в нашем `datadict` из листинга 14.9.

Рисунок 14.9 подчеркивает цель набора данных для проверки. Мы хотим, чтобы потери от оценки нашего набора данных для проверки были такими же хорошими, как наши потери проверки 1.939 из листинга 14.11. Если наши потери окажутся выше, это будет красным флагом того, что у нас, может быть, произошло переобучение! Давайте посмотрим, как наша модель работает, в следующем листинге.



Рис. 14.9. Набор отложенных данных: ответы на ранее не представленные вопросы с помощью нашего обученного режима

¹ В контексте машинного обучения и искусственного интеллекта это относится к процессу, когда часть данных удерживается от обучения модели для последующей проверки ее эффективности. – Прим. ред.

Листинг 14.12. Оценка обученной модели на отложенных примерах

```
evaluation = trainer.evaluate(eval_dataset=tokenized_
datasets["validation"])
display(evaluation)
```

Вывод:

```
{"eval_loss": 1.7851890325546265,
 "eval_runtime": 2.9417,
 "eval_samples_per_second": 5.099,
 "eval_steps_per_second": 0.34,
 "epoch": 3.0}
```

Значение `eval_loss` 1.785 от тестирования нашего отложенного набора проверки выглядит великолепно. Оно даже лучше, чем потери обучения и тестирования. Это означает, что наша модель работает хорошо и, скорее всего, не переобучает данные обучения или тестирования.

Вы можете продолжать обучение и улучшение модели, но мы продолжим с ней как с полностью обученной моделью, которую мы интегрируем в ридер для нашей системы вопрос–ответ.

14.4. Создание ридера с новой тонко настроенной моделью

Теперь, когда обучение модели нашего ридера завершено, мы интегрируем ее в конвейер вопрос–ответ, чтобы создать наш финальный ридер, который может извлекать ответы из вопросов и контекстов. Следующий листинг демонстрирует, как мы можем загрузить нашу модель в конвейер `question-answering`, предоставляемый библиотекой `transformers`.

Листинг 14.13. Загрузка тонко настроенной модели вопрос–ответ `outdoors`

```
device = get_processor_device()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
nlp2 = pipeline("question-answering", model=model_name,
               tokenizer=model_name, device=device)
```

Загрузив конвейер вопрос–ответ, мы извлечем некоторые ответы из некоторых пар вопрос–контекст. Давайте используем наш набор проверки из 10 документов, использованный ранее в разделе 14.3.4. Примеры проверки не использовались для обучения или тестирования модели, поэтому они должны стать хорошим лакмусовым тестом для того, насколько хорошо наша модель работает на практике.

В следующем листинге мы проверяем точность нашей модели вопрос–ответ на примерах набора проверки холдаута.

Листинг 14.14. Оценка тонко настроенной модели вопрос–ответ

```
def answer_questions(examples):
    answers = []
    success = 0
    for example in examples:
        question = {"question": example["question"][0],
                    "context": example["context"][0]}
        answer = nlp2(question)
        label = example["answers"][0]["text"][0]
        result = answer["answer"]
        print(question["question"])
        print("Label:", label)
        print("Result:", result)
        print("-----")
        success += 1 if label == result else 0
        answers.append(answer)
    print(f"{success}/{len(examples)} correct")

datadict["validation"].set_format(type="pandas", output_all_
columns=True)
validation_examples = [example for example in datadict["validation"]]
answer_questions(validation_examples)
```

Вывод:

```
How to get pine sap off my teeth
Label: Take a small amount of margarine and rub on the sap
Result: Take a small amount of margarine and rub on the sap
```

```
Why are backpack waist straps so long?
Label: The most backpacks have only one size for everyone
Result: The most backpacks have only one size for everyone
```

...

```
How efficient is the Altai skis "the Hok"?
Label: you can easily glide in one direction (forward) and if you try to
      glide backwards, the fur will "bristle up"
Result: you can easily go uphill, without (much) affecting forward gliding
        performance
```

7/10 Correct

Успешное извлечение 7 из 10 правильных ответов – впечатляющий результат. Поздравляем, теперь вы точно настроили LLM для реального варианта использования! Это завершает компонент reader нашей архитектуры, но нам все еще нужно объединить его с retriever, который находит начальные контексты-кандидаты для передачи в reader. В следующем разделе мы включим ретривер (нашу поисковую систему) для завершения сквозной системы вопрос–ответ.

14.5. Инкорпорация ретривера: использование модели вопрос–ответ с поисковым движком

Далее мы реализуем операцию реранкинга с использованием оценки уверенности ридера для ранжирования лучших ответов. Вот схема шагов, которые мы выполним в этом упражнении:

- 1 запрос индекса `outdoors` из поисковой коллекции, настроенной на высокий отзыв;
- 2 соединим наш вопрос с результатами топ- K документов и выведем ответы и оценки с помощью конвейера вывода NLP с ответами на вопросы;
- 3 переранжируем прогнозы ответов по убыванию оценки;
- 4 вернем правильный ответ и лучшие результаты, используя части, созданные на шагах 1–3.

Смотрите рис. 14.4 для освежения информации об этом потоке приложения.

14.5.1. Шаг 1: запрос ретривера

Наша цель на первом этапе извлечения – отзыв. В частности, какие все потенциально релевантные документы могут содержать наш ответ? Мы полагаемся на уже настроенную коллекцию поиска, чтобы получить этот отзыв, чтобы можно было передать качественные документы на этап реранкинга.

В следующем листинге реализована наша функция извлечения (ретривер), которая может принимать вопрос и возвращать начальный список релевантных документов для рассмотрения в качестве потенциальных контекстов для ответа.

Листинг 14.15. Ретривер, который ищет релевантные ответы

```
nlp = spacy.load("en_core_web_sm")
nlp.remove_pipe("ner")

def get_query_from_question(question):
    words = [token.text for token in nlp(question)
              if not (token.lex.is_stop or token.lex.is_punct)]
    return " ".join(words)

def retriever(question):
    contexts = {"id": [], "question": [], "context": [], "url": []}
    query = get_query_from_question(question)
    request = {"query": query,
               "query_fields": ["body"],
               "return_fields": ["id", "url", "body"],
               "filters": [("post_type", "answer")],
               "limit": 5}
    docs = outdoors_collection.search(**request)["docs"]
```

Использует английскую модель spaCy NLP.

Преобразует вопрос в запрос, удаляя стоп-слова и сосредотачиваясь на важных частях речи (см. блокнот для реализации).

Получает только документы с ответами (не вопросы).

```

for doc in docs:
    contexts["id"].append(doc["id"])
    contexts["url"].append(doc["url"])
    contexts["question"].append(question)
    contexts["context"].append(doc["body"])
return pandas.DataFrame(contexts)

example_contexts = retriever('What are minimalist shoes?')
display_contexts(example_contexts)

```

Ответ:

id	question	context
18376	What are minimalist shoes?	Minimalist shoes or "barefoot" shoes are shoes...
18370	What are minimalist shoes?	There was actually a project done on the defin...
16427	What are minimalist shoes?	One summer job, I needed shoes to walk on a ro...
18375	What are minimalist shoes?	The answer to this question will vary on your...
13540	What are minimalist shoes?	Barefoot Shoes Also known as minimalist shoes,...

Одной из проблем, с которой мы сталкиваемся при использовании вопроса в качестве запроса, является шум. Существует множество документов, в которых есть термины «кто», «что», «когда», «где», «почему» и «как», а также другие стоп-слова и менее важные части речи. Хотя BM25 может хорошо справляться с понижением приоритета этих терминов в функции ранжирования, мы знаем, что это не ключевые термины, которые ищет пользователь, поэтому мы удаляем их в функции `get_query_from_question`, чтобы уменьшить шум. Мы уже рассматривали маркировку частей речи с помощью `sraCu` ранее в главах 5 и 13, поэтому не будем повторять реализацию здесь (вы можете найти ее в блокноте).

Имея хороший набор документов, возвращенных поисковой системой, которые могут содержать ответы на вопросы пользователя, мы теперь можем передать эти документы в качестве контекстов в модель ридера.

14.5.2. Шаг 2: вывод ответов из модели ридера

Теперь мы можем использовать модель `reader` для вывода ответов на вопросы из каждого из N верхних контекстов. Листинг 14.16 реализует наш общий интерфейс ридера, который принимает выходные данные из `retriever` шага 1. Загрузка модели и конвейера для `retriever` следует тому же процессу, что и в листинге 14.13, в то время как остальная часть реализации `reader` специально обрабатывает генерацию возможных ответов (вместе с оценками для каждого ответа) из переданных контекстов.

Листинг 14.16. Ридер, который включает нашу тонко настроенную модель

```

from transformers import pipeline

device = get_processor_device()
model_name = "data/question-answering/roberta-base-squad2-fine-tuned"
qa_nlp = pipeline("question-answering", model=model_name,
                  tokenizer=model_name, device=device)

def reader(contexts):
    answers = []
    for _, row in contexts.iterrows():
        answer = qa_nlp({"question": row["question"],
                        "context": row["context"]})
        answer["id"] = row["id"]
        answer["url"] = row["url"]
        answers.append(answer)
    return answers

```

Создает конвейер spaCy с использованием нашей тонко настроенной модели.

Вызывает конвейер ридера для извлечения возможного ответа из каждого контекста.

Возвращает дополнительные метаданные о том, где был найден каждый ответ.

Ридер возвращает ответ из каждого контекста на основе нашей точно настроенной модели вместе с `id`, `url` и `score` ответа.

14.5.3. Шаг 3: реранкинг ответов

В листинге 14.17 показана простая функция, которая делает реранкинг ответов, просто сортируя их по оценке (выходные данные функции массы вероятности) из модели reader. Верхний в рейтинге ответ является наиболее вероятным и поэтому отображается первым. Вы можете показать один ответ или все, что возвращает ридер. Действительно, иногда может быть полезно предоставить задающему вопрос несколько вариантов и позволить ему сделать выбор. Это увеличивает вероятность отображения правильного ответа, но также занимает больше места в браузере или приложении, представляющем ответы, поэтому может потребоваться компромисс с пользовательским опытом.

Листинг 14.17. Реранкер делает сортировку соответственно оценкам ридера

```

def reranker(answers):
    return sorted(answers, key=lambda k: k["score"], reverse=True)

```

Следует отметить, что ваш реранкер может быть более сложным, потенциально включающим несколько условных моделей или даже пытающимся объединить несколько ответов вместе (например, перекрывающиеся ответы из нескольких контекстов). Для наших целей мы просто положимся на высшую оценку.

14.5.4. Шаг 4: возврат результатов путем объединения ретривера, ридера и реранкера

Теперь мы готовы собрать все компоненты нашей системы вопрос-ответ (QA, от англ. *Question-Answer*). Самая сложная часть сделана, поэтому мы можем поместить их в одну функцию, метко названную `ask`, которая будет принимать запрос и выводить ответ.

Листинг 14.18. Функция QA, объединяющая ретривер, ридер и реранкер

```
def ask(question):
    documents = retriever(question)
    answers = reader(documents)
    reranked = reranker(answers)
    print_answer(question, reranked)
ask('What is the best mosquito repellent?')
ask('How many miles can a person hike day?')
ask('How much water does a person need per day?')
```

Ответ:

```
What is the best mosquito repellent?
1116 DEET (0.606)
1056 thiamine (0.362)
569 Free-standing bug nets (0.158)
1076 Insect repellent is not 100% effective (0.057)
829 bear-spray (0.05)
```

```
How many miles can a person hike day?
17651 20-25 (0.324)
19609 12 miles (0.164)
19558 13 (0.073)
13030 25-35 (0.065)
4536 13 miles (0.022)
```

```
How much water does a person need per day?
1629 3 liters (0.46)
193 MINIMUM a gallon (0.235)
20634 0.4 to 0.6 L/day (0.207)
11679 4 litres (0.084)
11687 carry water (0.037)
```

Эти результаты выглядят довольно хорошо. Обратите внимание, что в некоторых случаях несколько контекстов могут возвращать один и тот же ответ. Как правило, это будет сильным сигналом правильного ответа, поэтому это может быть сигналом для рассмотрения интеграции в ваш повторный реранкинг.

Удивительно видеть качество результатов, возможных с использованием этих готовых моделей с минимальной переподготовкой. Респект сообществу NLP за то, что они сделали эти инструменты, мето-

ды, модели и наборы данных с открытым исходным кодом доступны и простыми в использовании!

Поздравляем, вы успешно внедрили сквозную систему вопрос–ответ, которая извлекает ответы из результатов поиска. Вы сгенерировали набор ответов silver, увидели, как улучшить их до набора golden, загрузили и настроили модель ридера вопрос–ответ и реализовали шаблон ретривер–ридер, используя вашу обученную модель и поисковую систему.

Однако с LLM мы можем сделать гораздо больше, чем просто извлекать ответы из результатов поиска. LLM можно настроить для получения сгенерированных (абстрактных) ответов на вопросы, чтобы генерировать ответы, невидимые в результатах поиска, но синтезированные из нескольких источников. Их также можно обучить суммировать результаты поиска для пользователей или даже синтезировать совершенно новый контент (текст, изображения и т. д.) в ответ на ввод пользователя. Многие LLM обучены на таком большом количестве данных в столь широком объеме человеческих знаний (например, большая часть известного интернета), что они часто могут выполнять широкий спектр задач, подобных этой, не отходя от кассы. Эти базовые модели, которые мы рассмотрим в следующей главе, прокладывают путь для следующей эволюции как ИИ, так и поиска на основе ИИ.

Резюме

- Система извлечения вопрос–ответ обычно следует шаблону ретривер–ридер, где возможные контексты (документы) находит ретривер и далее они анализируются с помощью модели ридера для извлечения наиболее вероятного ответа.
- Поисковая система служит отличным ретривером, поскольку она специально разработана для того, чтобы принимать запрос и возвращать ранжированные документы, которые, вероятно, будут служить релевантным контекстом для ответа на запрос.
- Модель ридера анализирует фрагменты текста, чтобы предсказать наиболее вероятное начало и конец ответа в каждом контексте, оценивая все варианты для извлечения наиболее вероятного ответа.
- Курирование набора данных для обучения занимает много времени, но вы можете автоматически сгенерировать набор silver данных для обучения с помощью предварительно обученной модели. Затем вы можете настроить ответы в наборе silver, чтобы сэкономить значительные усилия по сравнению с созданием всего набора golden данных для обучения вручную.
- Вы можете точно настроить предварительно обученную модель под свой конкретный набор данных, используя набор данных для обучения, тестирования и проверки на отложенных данных, а также оптимизируя функцию минимизации потерь.

15

Базовые модели и новые парадигмы поиска

В этой главе рассматривается:

- генерация с дополненной выборкой (RAG);
- генеративный поиск для обобщения результатов и абстрагированного ответа на вопросы;
- интеграция базовых моделей, оптимизация подсказок и оценка качества модели;
- генерация синтетических данных для обучения модели;
- реализация мультимодального и гибридного поиска;
- будущее поиска на основе ИИ.

Большие языковые модели (LLM), подобные тем, которые мы тестировали и настраивали в последних двух главах, были в авангарде достижений в области поиска на основе ИИ в последние годы. Вы уже видели некоторые из основных способов повышения качества поиска с помощью этих моделей, от улучшения интерпретации запросов и понимания документов путем сопоставления контента в эмбединги для плотного векторного поиска до помощи в извлечении ответов на вопросы из документов.

Но какие дополнительные передовые методы появляются на горизонте? В этой главе мы рассмотрим некоторые из последних достижений на стыке поиска и ИИ. Мы рассмотрим, как базовые модели используют

ся для расширения новых возможностей поиска на основе ИИ, таких как обобщение результатов, абстрактивные (сгенерированные) ответы на вопросы, мультимодальный поиск по типам носителей и даже диалоговые интерфейсы для поиска и извлечения информации. Мы рассмотрим основы новых парадигм поиска, таких как генеративный поиск, генерация, дополненная результатами поиска (RAG) и новые классы базовых моделей, переосмысливающих некоторые из способов, которыми мы вскоре приблизимся к передовой линии поиска на основе ИИ.

15.1. Что такое базовые модели

Базовая модель (модель фундамента, англ. *foundation model*) – это модель, которая предварительно обучена на большом количестве общих данных и разработана для того, чтобы быть в целом эффективной для широкого спектра задач. LLM – это подмножество базовых моделей, которые обучаются на очень большом количестве текста. Базовые модели также могут обучаться на изображениях, аудио или других источниках или даже на мультимодальных данных, включающих множество различных типов входных данных. На рис. 15.1 показаны общие категории базовых моделей.

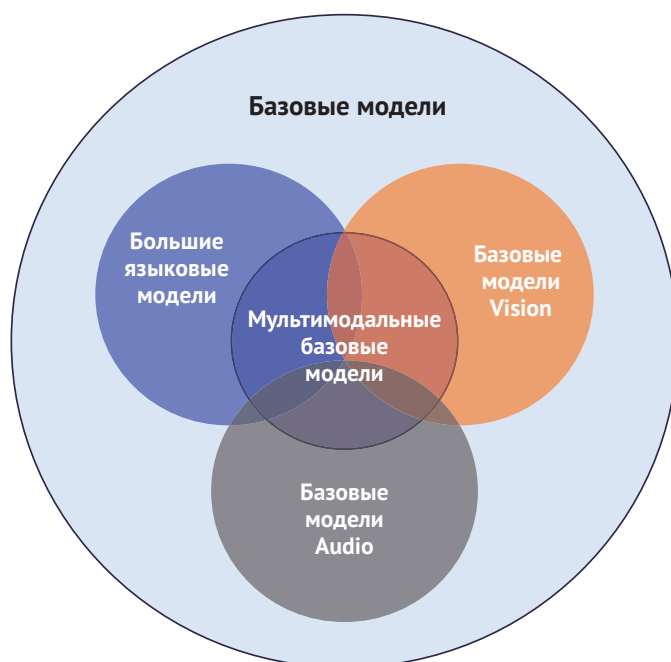


Рис. 15.1. Типы базовых моделей. LLM являются одним из нескольких типов базовых моделей

Базовая модель Vision может использоваться для маппинга изображений в эмбединги (например, как мы преобразовывали текст в эмбединги в главе 13), которые затем можно искать для включения поиска картинки по картинке.

Мультимодальная базовая модель может быть построена с использованием как текста, так и изображений (или других типов данных), и затем она может включать кросс-модальный поиск картинок на основе

текстовых запросов или текста на основе загруженных изображений в качестве запроса. Мы реализуем этот тип мультимодального поиска текста и изображений в разделе 15.3.2. Генеративные мультимодальные модели, такие как Stable Diffusion (модель преобразования текста в изображение), также могут использоваться для создания совершенно новых изображений на основе только текстовых подсказок. Мультимодальные базовые модели, которые могут обучаться как на изображениях, так и на тексте, также обычно называются *моделями языка визуализации* (VLM).

15.1.1. Что можно считать базовой моделью?

Базовые модели обычно обучаются на широком спектре данных, охватывающих множество тем, чтобы они были эффективны при обобщенной интерпретации и прогнозировании в разных областях. Эти модели называются «базовыми» моделями, потому что они могут служить базой (или основой), которую затем можно быстрее настроить на обучающих наборах, специфичных для конкретной области или задачи, чтобы лучше решать конкретные проблемы.

Базовые модели обычно соответствуют следующим критериям:

- 1 они *большие*, обычно обучаются на огромных объемах данных, часто с миллиардами или триллионами параметров;
- 2 они *предварительно обучены*, используя значительную вычислительную мощность для получения весов модели, которые можно сохранить и развернуть (или настроить) позже;
- 3 они *обобщаемы* для многих задач, в отличие от моделей, ограниченных конкретными задачами;
- 4 они *адаптируемы*, используют подсказки для извлечения дополнительного контекста из своей обученной модели для корректировки своего прогнозируемого вывода. Это делает типы запросов, которые они могут принимать, очень гибкими;
- 5 они *самоконтролируемы*, автоматически обучаясь на основе необработанных данных тому, как соотносить и интерпретировать данные и представлять их для будущего использования.

Мы уже работали с несколькими базовыми моделями в предыдущих главах, включая BERT, которая является одной из самых ранних базовых моделей, и RoBERTa, которую мы использовали в главе 13 для генерации эмбедингов и выполнения семантического поиска по этим эмбедингам. Модели Sentence Transformer, такие как SBERT (Sentence-BERT) и SROBERTa (Sentence-RoBERTa), являются моделями, которые были тонко настроены на основе базовых моделей BERT и RoBERTa для достижения превосходных результатов в задаче семантического текстового сходства (STS). Мы также тонко настроили модель `deepset/goberta-base-squad2` в главе 14; это модель, основанная на базовой модели RoBERTa, которая была тонко настроена для задачи вопрос-ответ. Технически SBERT, SROBERTa и `deepset/goberta-base-squad2` сами по себе являются тонко настроенными базовыми моделями, которые могут быть в дальнейшем использованы в качестве основы для более тонкой настройки для создания дополнительных моделей.

Доминирующей архитектурой для базовых моделей в настоящее время является модель трансформера, хотя рекуррентные нейронные сети (использующие архитектуры, такие как МАМВА) также могут использоваться, и дополнительные архитектуры неизбежно будут развиваться с течением времени. Большинство моделей на основе трансформеров могут использоваться для генерации векторов эмбединга или предиктивного вывода.

Сила ответов базовых моделей отражает качество трех процессов: обучения, тонкой настройки и подсказки.

15.1.2. Обучение, тонкая настройка в сравнении с подсказкой

Обучение (или *предварительное обучение*, англ. *pretraining*) – это процесс, при котором огромный объем данных (часто большая часть интернета) используется для изучения весов модели для миллиардов или триллионов параметров в глубокой нейронной сети базовой модели. Этот процесс иногда может быть очень дорогим, занимать месяцы и стоить миллионы долларов из-за требований к вычислениям и энергии. Этот процесс позволяет выполнить сжатие с потерями большей части человеческих знаний в нейронную сеть, из которой факты и отношения (слова, лингвистика, ассоциации и т. д.) могут быть распакованы позже. Вспомним из раздела 13.3, что обучение трансформера на тексте обычно следует за самоконтролируемым процессом обучения, который оптимизируется для прогнозирования замаскированных токенов в текстовых последовательностях для измерения общего понимания текста (тест Cloze, описанный в разделе 13.3.1). Это обучение может включать любые конкретные наборы данных, которые могут быть полезны для базы знаний модели, такие как компьютерный код или контент, специфичный для предметной области (финансовые документы, научные статьи, иностранные языки, мультимодальный контент и т. д.).

Тонкая настройка – это процесс, посредством которого базовая модель может быть снабжена дополнительными данными или инструкциями, специфичными для предметной области. Например, если вы хотите, чтобы модель следовала инструкциям или действовала как человек или чат-бот, вы можете обучить модель с ответами на ввод, которые отражают это поведение. Существует несколько подходов к тонкой настройке в зависимости от задачи или архитектуры, а также вычислительных и бюджетных требований. Некоторые типы тонкой настройки изменяют все веса модели, что полезно, если настройка домена должна быть очень конкретной, а задача исходной модели та же. Более эффективные или специфичные для задачи подходы могут оставить все существующие веса нетронутыми и добавить один или несколько дополнительных слоев в нейронную сеть базовой модели. Тонкая настройка позволяет этим новым источникам расширять возможности исходной модели для новых данных или шаблонов с гораздо менее масштабным процессом обучения, сосредоточенным на конкретных данных или целях.

Создание подсказок – это процесс подачи входных данных в обученную языковую модель для получения выходных данных. Подсказка –

это последний шаг «обучения» для тонкой настройки модели, но он происходит во время вывода, а не во время обучения. Мы можем предоставить столько контекста в подсказке, сколько позволяет модель, что означает, что подсказкой можно манипулировать, чтобы использовать этот дополнительный контекст для воздействия на вывод. Например, рассмотрим вывод запросов в листингах 15.1 и 15.2, когда они отправляются в языковую модель GPT-4 OpenAI.

Листинг 15.1. Запрос без инжиниринга подсказок¹

Query: "What is a unicorn?"

Response:

A unicorn is a mythical creature that is typically depicted as a horse with a single, spiraled horn projecting from its forehead. Unicorns have been a part of human mythology and folklore for centuries and are often associated with purity, grace, and magic. They are said to possess healing powers and to be able to bring good luck to those who encounter them. While unicorns are not real animals, they continue to be a popular symbol in modern culture, appearing in literature, art, and popular media.

(Запрос: «Что такое единорог?»)

Ответ: Единорог – мифическое существо, которое обычно изображается как лошадь с одним спиральным рогом, торчащим из лба. Единороги были частью человеческой мифологии и фольклора на протяжении веков и часто ассоциировались с чистотой, грацией и магией. Говорят, что они обладают целительной силой и могут приносить удачу тем, кто их встречает. Хотя единороги не являются реальными животными, они продолжают оставаться популярным символом в современной культуре, появляясь в литературе, искусстве и популярных СМИ.)

Листинг 15.1 дает хорошее общее описание единорога. Сравните это с листингом 15.2, который использует проектирование подсказок, чтобы дать базовой модели персону чат-бота под названием AIPS Chat, а затем говорит базовой модели отвечать так, как будто у нее есть докторская степень по биологии.

Листинг 15.2. Запрос с дополнительным контекстом из инжиниринга подсказок

Query: "You are a chatbot named AIPS Chat.
You have a Ph.D. in biology.
What is a unicorn?"

¹ Программная инженерия – это проектирование, написание, тестирование и курирование компьютерной программы с целью решить задачи многих пользователей. Речь идет о создании надежных и безопасных решений, которые выдержат проверку временем и будут работать для некоторых возможно заранее неизвестных задач, помимо очевидных. – *Прим. ред*

Response:

Hello! I am a chatbot named AIPS Chat, and I have a Ph.D. in biology. While

unicorns have been depicted in various cultures throughout history, they are not real animals that exist in the natural world.

From a biological perspective, a unicorn can be described as a mythical creature resembling a horse with a single, spiraled horn on its forehead.

However, since unicorns are purely fictional, there is no actual scientific data or research that can be conducted on them.

It is important to note that while many animals possess unique characteristics, such as the horn of a rhinoceros or the tusk of a narwhal, these features are not magical and do not provide any special abilities or powers to the animal possessing them.

In conclusion, while unicorns are a fascinating aspect of human folklore and mythology, they do not have a basis in reality within the field of biology.

(Запрос: «Вы чат-бот под названием AIPS Chat.

У вас есть докторская степень по биологии.
Что такое единорог?»

Ответ:

Здравствуйте! Я чат-бот под названием AIPS Chat, и у меня есть докторская степень по биологии. Хотя единороги изображались в разных культурах на протяжении всей истории, они не являются реальными животными, существующими в естественном мире.

С биологической точки зрения единорога можно описать как мифическое существо, напоминающее лошадь с одним спиральным рогом на лбу. Однако, поскольку единороги являются чисто вымышленными существами, нет никаких фактических научных данных или исследований, которые можно было бы провести по ним. Важно отметить, что, хотя многие животные обладают уникальными характеристиками, такими как рог носорога или бивень нарвала, эти особенности не являются магическими и не дают никаких особых способностей или сил животному, обладающему ими.

В заключение следует отметить, что, хотя единороги являются увлекательным аспектом человеческого фольклора и мифологии, они не имеют под собой реальной основы в области биологии.)

Запрос в листинге 15.2, который включает контекст чат-бота, имеющего докторскую степень по биологии, использует этот дополнительный контекст для информирования своего ответа. Если бы у нас был доступ для чтения-записи к языковой модели, мы могли бы точно настроить ее с помощью входных данных и ответов, сгенерированных докторской степенью по биологии и чат-ботом. В листинге мы смогли достичь аналогичного результата, просто предоставив модели подсказку для извлечения этого контекста из ее уже обученной модели. Тонкая настройка обычно генерирует лучший ответ на такие вопросы, но подсказки гораздо более гибки, поскольку с их помощью вы мо-

жете предоставить любой контекст во время вывода, тогда как тонкая настройка будет более подходящей для общих признаков, которые модель должна изучить и представить во всех будущих взаимодействиях, если вы хотите именно этого.

Предоставление дополнительного контекста в подсказке может быть ключом к получению наилучшего результата, который может представить базовая модель. Поскольку генеративные базовые модели последовательно предсказывают следующий токен в последовательности, по одному токenu за раз, принуждение модели выводить более релевантный контекст в ответе может привести к тому, что модель будет генерировать более релевантный вывод. Большая предварительно обученная модель способна к *обучению с небольшим количеством примеров* новых классов (умеет обучаться в контексте без дальнейшего обучения, предоставляя два или три примера с подсказкой). Например, в листинге 15.2 мы увидели, что при добавлении контекста «биология» к подсказке ответ после этого включал такие фразы, как «в естественном мире», «с биологической точки зрения», «фактические научные данные или исследования» и «в области биологии». Самая сложная и дорогая часть обучения – это первоначальное создание базовой модели. После обучения тонкая настройка для конкретных задач выполняется относительно быстро и недорого и часто может быть выполнена с помощью достаточно небольшого обучающего набора. Если мы сравним обучение этих моделей с обучением человека, базовая модель может быть похожа на ученика средней школы, который в целом умеет читать и писать и может ответить на основные вопросы по математике, науке и мировой истории. Если мы хотим, чтобы студент мог создавать и интерпретировать финансовые отчеты (отчеты о прибылях и убытках, отчеты о движении денежных средств и балансы), ему нужно будет пройти полный и долгий курс бухгалтерского учета, чтобы освоить эти навыки. Однако после 18 или более лет обучения (жизненный опыт и школа) студент, скорее всего, сможет изучить бухгалтерский учет в течение нескольких месяцев, чтобы создавать и интерпретировать финансовые отчеты. Аналогично с базовыми моделями начальная фаза обучения занимает больше всего времени и обеспечивает базу, на которой можно гораздо быстрее освоить дальнейшие знания и навыки.

15.2. Генеративный поиск

Большая часть нашего пути к созданию поиска на основе ИИ была сосредоточена на поиске результатов, ответов или действий, соответствующих намерениям пользователя. В главе 14 мы зашли так далеко, что извлекли конкретные ответы на вопросы, используя наш шаблон ретривера-ридера. Но что, если вместо возврата реальных документов или извлеченных ответов наша поисковая система могла бы генерировать новый контент в ответ на запросы на лету?

Генеративные модели – это модели, которые могут генерировать новый контент на основе входящих подсказок. Их выход может быть

текстовым контентом, изображениями, сгенерированными из текстового ввода, аудио, эмулирующими определенных людей или звуки, или видео, объединяющими как аудио, так и видео. Текст может быть сгенерирован для описания изображения, или – в качестве альтернативы – изображение, аудио или видео могут быть сгенерированы для описания текста в другой модальности.

Мы вступаем в мир, где кто-то сможет ввести поисковый запрос, а поисковая система может вернуть полностью выдуманный контент и изображения, сгенерированные на лету, в ответ на запрос или подсказки пользователя. Хотя это может быть удивительно с запросами типа

What is an optimal agenda for three days in Paris assuming I really like fine dining, historic buildings and museums, but hate crowds

(«Какова оптимальная программа на три дня в Париже, если я действительно люблю изысканную кухню, исторические здания и музеи, но ненавижу толпы»),

это также вводит серьезные этические соображения.

Должна ли поисковая система нести ответственность за синтез информации для своих пользователей, вместо того чтобы возвращать существующий контент для их интерпретации? Что, если поисковая система политически или коммерчески предвзята и пытается повлиять на мышление или поведение пользователей? Что, если поисковая система используется как инструмент цензуры или для распространения пропаганды (например, изображений или текста, которые были изменены на лету), чтобы обмануть людей и заставить их принять ложные убеждения или предпринять опасные действия?

Некоторые из наиболее распространенных вариантов использования генеративного поиска – это сгенерированные ответы на вопросы и обобщение результатов. В то время как традиционные подходы к поиску возвращали «десять синих ссылок» или предопределенные информационные поля с информацией, соответствующей определенным запросам, генеративный поиск фокусируется на создании поисковых ответов на лету на основе динамического анализа результатов поиска. Слева направо на рис. 15.2 показан переход от этих традиционных подходов поиска к генеративному поиску.

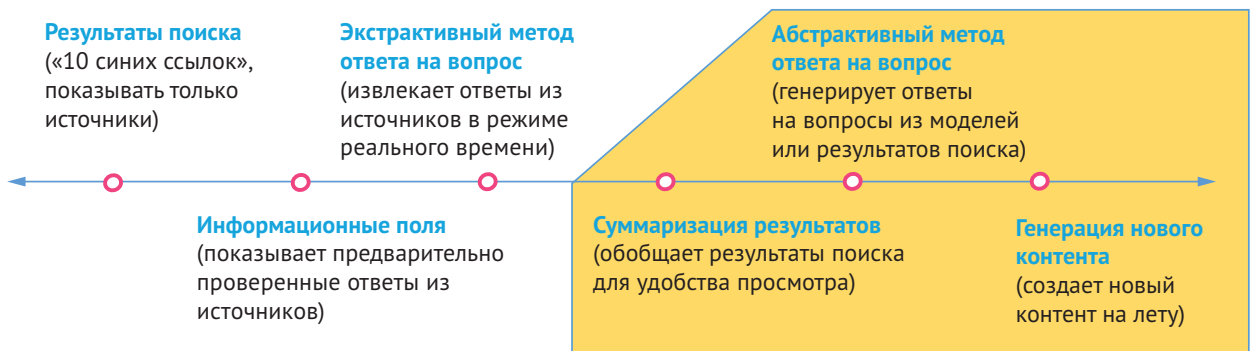


Рис. 15.2. Спектр перехода традиционных методов поиска (слева) к генеративному поиску (справа)

Экстрактивный метод ответа на вопрос, описанный в главе 14, начинает движение к генеративному поиску, поскольку он анализирует результаты поиска, чтобы возвращать прямые ответы на вопросы, а не только результаты поиска или предопределенные ответы. Однако это не совсем «генеративный» поиск, поскольку он по-прежнему возвращает только ответы, точно такие же, как представлены в искомых документах, без какого-либо дополнительного синтеза или генерации нового контента.

По мере продвижения вправо на рис. 15.2 суммирование результатов является первой техникой, которую можно полностью считать генеративным поиском. Суммаризация результатов, которое мы рассмотрим в разделе 15.2.2, представляет собой процесс взятия результатов поиска и резюмирования их содержания. Это может быть весьма полезно для пользователя, особенно если он исследует тему и у него нет времени читать все результаты. Вместо того чтобы пользователю нужно было кликать на несколько ссылок и читать и анализировать каждый из результатов, можно вернуть резюме страниц (вместе с цитатами, если необходимо), что экономит время пользователя на оценку содержания.

Абстрактивный метод ответа на вопрос очень похож на экстрактивный в том, что он пытается ответить на вопрос пользователя, но делает это либо путем анализа результатов поиска (как при резюмировании результатов), либо просто полагаясь на базовую модель для генерации ответа. Опора на базовую модель с большей вероятностью приведет к выдуманным или галлюцинаторным результатам, поэтому часто бывает полезно использовать результаты поиска как часть контекста подсказки для обучения в контексте. Мы рассмотрим этот процесс генерации, дополненной результатами поиска, в следующем разделе. Многие крупные поисковые системы и стартапы интегрировали эти генеративные модели и интерактивные сеансы чата в свои поисковые возможности, что обеспечивает моделям доступ к интернету (или, по крайней мере, к копии интернета в форме поискового индекса). Это позволяет моделям иметь представление о мировой информации практически в реальном времени. Это означает, что модели могут знать подробную общедоступную информацию о пользователях, с которыми они взаимодействуют, что может формировать результаты, а также означает, что любой может делать вложения вредоносной информации или намерений в эти модели, создавая веб-страницы, которые связывают понятия вводящим в заблуждение образом. Если мы не будем осторожны, область поисковой оптимизации (SEO) может перейти от попыток поднять веб-сайт выше в результатах поиска к попыткам манипулировать ИИ для предоставления вредоносных ответов конечным пользователям. Применение навыков критического мышления и проверка вывода этих моделей будут иметь важное значение для любого, кто их использует, но, к сожалению, модели могут быть настолько убедительными, что многие люди, вероятно, будут обмануты и поверят вводящим в заблуждение выводам, если не будут приняты существенные меры предосторожности. Эти модели будут продолжать развиваться, чтобы помогать поисковикам, интерпре-

тируя результаты поиска. Хотя многие люди могут мечтать о личном помощнике на основе ИИ, этические соображения, связанные с тем, чтобы ИИ генерировал знания, которые вы потребляете ежедневно, должны быть рассмотрены с осторожностью.

15.2.1. Генерация с дополненной выборкой

Рабочий процесс использования поисковой системы или векторной базы данных для поиска соответствующих документов, которые могут быть предоставлены в качестве контекста для LLM, обычно называется *генерацией с аугментацией*¹, или *генерацией с дополненной выборкой*².

Мы уже несколько раз упоминали RAG, и это большое модное слово сейчас, когда данная книга публикуется. Для этого есть очень веская причина: обучение языковых моделей является формой сжатия с потерями обучающих данных, и модели не могут добросовестно хранить огромный объем данных, на которых они были обучены, не теряя точности. Их информация также датируется последним временем их обучения – моделям сложно принимать решения об изменяющейся информации без какого-либо постоянно обновляемого внешнего источника данных.

Независимо от того, сколько контекста языковая модель может принять в своей подсказке, вычислительно нецелесообразно или, по крайней мере, невыгодно полагаться на LLM как на источник истины для всей информации, на которой она была обучена. К счастью, вся цель поисковых систем – найти релевантный контекст для любого входящего запроса. Фактически *вся эта книга* – глубокое погружение в часть *извлечения данных* RAG. Обсуждая в этой главе обобщение результатов, сгенерированные ответы на вопросы и генеративный поиск, мы затрагиваем *генеративной* частью RAG.

Для выполнения RAG большинство современных библиотек берут документы, разбивают их на один или несколько фрагментов (чанков), а затем индексируют эмбединги для каждого фрагмента в поисковой системе или векторной базе данных. Затем во время генерации приложение, запрашивающее языковую модель, создает запросы для поиска дополнительной информации, необходимой для выполнения следующего запроса, генерирует эмбединги для этих запросов, выполняет поиск по плотным векторам для поиска разделов с наивысшей оценкой, используя сходство векторов (обычно скалярное произведение), и передает полученные ранжированные разделы вместе с запросом в генеративную модель ИИ.

¹ Аугментация в программировании – это техника, которая позволяет искусственно увеличить объем данных, генерируя новые точки из существующих. Это достигается путем внесения небольших изменений в данные или использования моделей машинного обучения для генерации новых точек. – *Прим. ред.*

² Генерация с дополненной выборкой (Retrieval-Augmented Generation, RAG) – это метод в области искусственного интеллекта, который сочетает генерацию текста с извлечением информации из внешних источников. Этот подход позволяет моделям ИИ получать актуальные данные и использовать их для создания более точных и релевантных ответов. – *Прим. ред.*

Проблемы с чанкингом

Этот процесс разделения документов на фрагменты известен как *чанкинг* (chunking), и он одновременно необходим и проблематичен. Проблему можно понимать как соперничество между тремя ограничениями.

- Ограничения векторной базы данных – многие векторные базы данных предназначены для индексации отдельных векторов, но суммирование всего документа в один вектор может привести к потере значительной части специфичности и контекста документа. RAG объединяет эмбединг по всему документу в более неопределенный сводный эмбединг.
- Потеря контекста между независимыми фрагментами – разделение документа на множество отдельных документов для сохранения специфики каждого раздела (главы, абзаца, предложения, слова или другого полезного семантического чанка) может привести к потере контекста между чанками. Если вы разделите документ на 10 чанков и независимо сгенерируете эмбединги, общий контекст между этими эмбедингами будет потерян.
- Вычислительная сложность работы с множеством чанков – чем больше у вас фрагментов, тем больше векторов вам нужно индексировать и тем больше вам нужно искать. Это может оказаться вычислительно затратным и медленным, и также может быть сложно управлять релевантностью результатов между чанками, когда вам приходится взвешивать много совпадений в одном и том же исходном документе относительно меньшего количества, но лучших совпадений в других документах.

На ранних этапах реализации RAG для генеративного ИИ многие люди сосредоточились на разделении документов на несколько отдельных документов, чтобы преодолеть ограничения векторной базы данных. Сложность вычислений можно контролировать с помощью хороших алгоритмов ANN (приблизительный ближайший сосед), но потеря контекста между чанками в этом случае все еще проблематична, и откровенно расточительно создавать неограниченное количество перекрывающихся чанков вместо использования лучших алгоритмов для моделирования эмбедингов.

В качестве альтернативного подхода несколько поисковых систем и векторных баз данных (таких как Vespa и Qdrant) уже поддерживают многозначные векторные поля, что позволяет создавать произвольное количество чанков, а также создавать перекрывающиеся чанки в одном документе (так что одно предложение или абзац могут быть частью нескольких чанков, тем самым мы сохраняем больше контекста между чанками). Такая многовекторная поддержка, вероятно, станет стандартной в ближайшие годы для поддержки появляющихся контекстуализированных подходов позднего взаимодействия, таких как те, которые представлены в семействе моделей ColBERT.

Будущее RAG

RAG как дисциплина все еще находится в зачаточном состоянии, но она быстро развивается. Для выполнения RAG было разработано множество библиотек и фреймворков, но они часто основаны на слишком простом понимании поиска информации. Текущие реализации RAG, как правило, полностью полагаются на языковую модель и сходство векторов для релевантности, игнорируя большинство других методов поиска на основе ИИ, которые мы рассмотрели в этой книге. В контексте измерений намерения пользователя (рис. 1.7) это имеет три эффекта:

- контекст контента обрабатывается хорошо понятийно (предполагая, что выбранная модель эмбединга была обучена для поиска документа запроса), но не на основе конкретных ключевых слов (из-за области действия векторов, суммирующих несколько ключевых слов);
- контекст домена обрабатывается, а языковая модель настраивается;
- контекст пользователя обычно полностью игнорируется.

Для измерения понимания контента намерения пользователя (см. раздел 1.2.5) развиваются перспективные подходы, которые используют контекстуализированное позднее взаимодействие с эмбедингами, и они могут в конечном итоге полностью преодолеть необходимость в разбиении на чанки. Эти подходы (такие как ColBERT, ColBERTv2, ColPali и их последователи) включают в себя генерацию эмбединга на токен в документе, но где эмбединг каждого токена использует контекст этого токена в пределах всего документа (или, по крайней мере, длинный окружающий контекст). Это предотвращает потерю контекста между чанками и позволяет избежать необходимости индексировать неограниченное количество чанков в движок. Такие подходы имеют гораздо больше смысла для RAG и поиска в целом, чем некоторые из более наивных подходов, упомянутых в последнем подразделе. Мы ожидаем, что аналогичные подходы будут развиваться в ближайшие годы и значительно улучшат отзыв по сравнению с текущими базовыми подходами ранжирования.

Концепции поиска, которые вы изучили в этой книге, касающиеся измерений намерений пользователя, отраженного интеллекта и моделей сигналов, семантических графов знаний, обучения ранжированию и циклов обратной связи (в сочетании с векторным поиском на основе LLM), на световые годы опережают большинство текущих реализаций RAG, которые используют только подмножество их.

Тем не менее будущее RAG светлое, и, вероятно, в ближайшие несколько лет появятся более сложные подходы к решению текущих проблем и будут реализованы лучшие методы поиска и ранжирования в процессе RAG. Методы RAG быстро меняются и развиваются, и эта область созрела для постоянных инноваций и улучшений.

В следующем разделе мы рассмотрим один из самых популярных методов генеративного поиска с использованием RAG: суммаризацию результатов (с цитатами).

15.2.2. Суммаризация результатов с использованием базовых моделей

В главе 3 мы заявили, что поисковая система в первую очередь отвечает за три вещи: индексирование, сопоставление и ранжирование результатов. Глава 13 уже показала, как улучшить индексацию (с эмбедингами), сопоставление (с приблизительным поиском ближайшего соседа) и ранжирование результатов (со сравнением сходства на векторах эмбединга). Но то, как мы возвращаем результаты, часто так же важно, как и сами результаты.

В главе 14 мы продемонстрировали, как извлекать ответы из ранжированных документов, содержащих эти ответы, что может быть радикальным улучшением по сравнению с возвратом полных документов и принуждением пользователей анализировать их по отдельности. Но что, если ответ на вопрос требует анализа документов? Что, если желаемый ответ на самом деле является результатом анализа, который объединяет информацию из нескольких документов в единый, хорошо обоснованный ответ?

Одной из проблем с ответами, сгенерированными непосредственно из LLM, является то, что они создаются на основе статистических распределений вероятностей из параметров, изученных моделью. Хотя модель могла быть обучена на огромном количестве источников данных, она не хранит эти источники данных напрямую. Вместо этого модель сжимается в представление данных с потерями. Это означает, что вы не можете рассчитывать на то, что вывод LLM будет точно отражать входные данные – только их приближение.

В результате, как известно, фундаментальные модели галлюцинируют – генерируют ответы, которые составляют неверные факты или искажают темы в ответе. Контекст, предоставленный в подсказке, также сильно влияет на ответ, до такой степени, что фундаментальные модели иногда могут быть скорее отражением выбора слов и предвзятости пользователя, чем законным ответом на вопрос. Хотя это может быть полезно в творческих начинаниях, это делает сегодняшние фундаментальные модели довольно ненадежными для безопасного создания основанных на фактах ответов, на которые исторически полагаются поисковые системы. Чтобы пользователи действительно доверяли результатам, они должны иметь возможность проверять источники информации. К счастью, вместо того чтобы напрямую полагаться на фундаментальные модели для ответов на вопросы, мы можем объединить поисковую систему с фундаментальной моделью, используя RAG для создания гибридного вывода.

Использование таких фундаментальных моделей для обобщения результатов поиска – отличный способ наполнить вашу поисковую систему лучшими ответами на основе ИИ. Давайте рассмотрим пример выполнения резюмирования результатов поиска с цитатами с использованием базовой модели. Мы будем использовать вывод из модели GPT-4 OpenAI, но вы можете получить аналогичный вывод из большинства текущих открытых исходных кодов или разрешенных лицензированных LLM.

Рабочий процесс включает два шага:

- 1 выполнение поиска. Это может включать в себя преобразование запроса в эмбединг и выполнение плотного векторного поиска или использование любой другой техники, которую мы обсудили, чтобы найти наиболее релевантные результаты;
- 2 создание подсказки, инструктирующей вашу базовую модель принять запрос пользователя и прочитать и суммаризировать набор результатов поиска, возвращенных из запроса.

В следующем листинге показан пример подсказки, объединяющей результаты поиска из шага 1 в подсказку из шага 2.

Листинг 15.3. Подсказка базовой модели для суммаризации результатов поиска

Web search results:

[1] "A large language model, or LLM, is a deep learning algorithm that can recognize, summarize, translate, predict and generate text and other content based on knowledge gained from massive datasets."

URL: <https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/>

[2] A large language model (LLM) is a language model consisting of a neural network with many parameters (typically billions of weights or more), trained on large quantities of unlabeled text using self-supervised learning. LLMs emerged around 2018 and perform well at a wide variety of tasks. This has shifted the focus of natural language processing research away from the previous paradigm of training specialized supervised models for specific tasks.

URL: https://en.wikipedia.org/wiki/Large_language_model

[3] "In this guide, we'll discuss everything you need to know about Large Language Models (LLMs), including key terms, algorithms, fine-tuning, and more. As ChatGPT has taken the internet by storm crossing

1 million users in its first 5 days, you may be wondering what machine learning algorithm is running under the hood. While ChatGPT uses a specific type of reinforcement learning called "Reinforcement Learning from Human Feedback (RLHF)", at a high level it is an example of a Large Language Model (LLM)."

URL: <https://www.mlq.ai/what-is-a-large-language-model-llm/>

Instructions: Using the provided web search results, write a comprehensive reply to the given query. Make sure to cite results using [[number](URL)] notation after the reference. If the provided search results refer to multiple subjects with the same name, write separate answers for each subject.

Query: What is a large language model?

Когда весь листинг 15.3 передается как подсказка в языковую модель, мы получаем следующий результат.

Листинг 15.4. Ответ на подсказку суммаризации из листинга 15.3

```
A large language model (LLM) is a deep learning algorithm that can recognize, summarize, translate, predict, and generate text and other content based on knowledge gained from massive datasets [[1](https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/)]. It consists of a neural network with many parameters (typically billions of weights or more) and is trained on large quantities of unlabeled text using self-supervised learning [[2](https://en.wikipedia.org/wiki/Large_language_model)]. LLMs emerged around 2018 and have performed well at a wide variety of tasks, shifting the focus of natural language processing research away from the previous paradigm of training specialized supervised models for specific tasks [[2](https://en.wikipedia.org/wiki/Large_language_model)]. One example of a LLM is ChatGPT, which uses a specific type of reinforcement learning called "Reinforcement Learning from Human Feedback (RLHF)" [[3](https://www.mlq.ai/what-is-a-large-language-model-llm/)].
```

Этот вывод извлекает критически важные детали из результатов поиска для ответа на исходный запрос – What is a large language model? (Что такое большая языковая модель?), но делает это таким образом, что цитирует исходные статьи. Если этот ответ слишком многословен, можно добавить дополнительное уточнение к подсказке в конце инструкций: Be concise (Будьте краткими). Это приводит к следующему выводу вместо этого:

...
Query: What is a large language model? Be concise.

Листинг 15.5. Результаты для подсказки листинга 15.3 с Be concise

```
A large language model (LLM) is a deep learning algorithm that uses neural networks with billions of parameters, trained on massive unlabeled text datasets for various language tasks [[2](https://en.wikipedia.org/wiki/Large_language_model)][[1](https://blogs.nvidia.com/blog/2023/01/26/what-are-large-language-models-used-for/)].
```

Многие поисковые системы начали полагаться на базовые модели для интерпретации и цитирования результатов поиска, подобных этим. Добавляя дополнительные инструкции, такие как «избегайте расплывчатости, противоречивости или сообщений не по теме», «используйте беспристрастный и журналистский тон» или даже «пишите на уровне чтения ученика пятого класса», вы можете настроить качество и тон резюме в соответствии с потребностями вашей поисковой системы.

15.2.3. Генерация данных с использованием базовых моделей

Помимо базовых моделей, добавляющих слой синтеза и резюмирования поверх результатов поиска, одним из других новых вариантов использования моделей является генерация синтетических обучающих данных, специфичных для предметной области и задач.

Если вспомнить многочисленные методы поиска на основе ИИ, которые мы уже изучили, то многие из них требуют существенных обучающих данных для реализации:

- модели бустинга сигналов требуют данные пользовательских сигналов, показывающие, какие запросы пользователя каким документам соответствуют;
- модели кликов, полезные для автоматизированного LTR, требуют понимания того, на какие документы пользователи кликают и какие документы пользователи пропускают в своих результатах поиска;
- семантические графы знаний требуют индекса данных для поиска связанных терминов и фраз.

Но что, если мы настроим базовую модель так, чтобы «придумать документы об отсутствующих темах» или «придумать реалистичные запросы», связанные с каждым документом? Или, что еще лучше, что, если нам вообще не нужно настраиваться, а вместо этого можно построить подсказку для генерации отличных запросов для документов? Такие данные можно использовать для генерации синтетических сигналов, чтобы улучшить релевантность.

Листинги 15.6–15.9 показывают, как мы можем использовать подсказку для поиска запросов и оценок релевантности, объединяя LLM с поиском в нашей коллекции Stack Exchange outdoors из главы 14.

Для нашего упражнения мы обнаружили, что лучше дать подсказке несколько документов, а не только один и еще лучше вернуть связанные документы (из той же темы или списка результатов из предыдущего запроса). Это важно, потому что в каждом наборе документов есть нишевые темы, поэтому наличие связанных документов помогает отвечать на более детальные запросы вместо общих. Кроме того, хотя все документы могут быть не совсем релевантны исходному запросу (или могут содержать много шума), это все равно дает LLM шансы на понимание нашего корпуса, в отличие от базирования его контекста на одном примере. Следующий листинг показывает шаблон, используемый с LLM для генерации запросов-кандидатов для списка документов.

Листинг 15.6. Подсказка, которая генерирует запросы, описывающие документы

```
What are some queries that should find the following documents? List at least 5 unique queries, where these documents are better than others in an outdoors question and answer dataset. Be concise and only output the list of queries, and a result number in the format [n] for the best result in the resultset. Don't print a relevance summary at the end.
```

```
### Results:
{resultset}
```

Следующий листинг показывает код, который может генерировать {resultset} для вложения в подсказку из листинга 15.6. С примером запроса *what are minimalist shoes?* мы получаем набор resultset с помощью функции *retriever* из листинга 14.15. Ретривер предоставляет контексты ответа для запроса.

Листинг 15.7. Создание результатов текстового поиска, удобных для подсказок

```
example_contexts = retriever("What are minimalist shoes?")
resultset = [f"{idx}. {ctx}" for idx, ctx
             in enumerate(list(example_contexts[0:5]["context"]))]
print("\n".join(resultset))
```

Вывод:

Нам нужна только информация об индексе и контексте из первых 5 результатов, возвращенных ретривером, и мы добавляем к каждому результату его индекс от 0 до 4.

0. Minimalist shoes or "barefoot" shoes are shoes that provide your feet with some form of protection, but get you as close to a barefoot experience as possible...
1. There was actually a project done on the definition of what a minimalist shoe is and the result was "Footwear providing minimal interference with the natural movement of the foot due to its high flexibility, low heel to toe drop, weight and stack height, and the absence of motion control and stability devices". If you are looking for a simpler definition, this is what Wikipedia says, Minimalist shoes are shoes intended to closely approximate barefoot running conditions...
2. One summer job, I needed shoes to walk on a rocky beach, sometimes in the water, for 5 to 10 miles per day all summer. Stretchy neoprene shoes were terrible for this- no traction and no support. So I used regular sneakers. I chose a pair of cross country racing flats... The uppers were extremely well ventilated polyester, so they drained very quickly, and the thin material dried much faster than a padded sandal, and certainly much faster than a regular sneaker of leather or cotton would... The thing to look for is thin fabric that attaches directly to the sole, with no rubber rim that would keep the water from draining...
3. ... It's not unhealthy to wear minimalist footwear, but on what terrain your wear them could be bad for your body in the long run. Human beings were never meant to walk or run exclusively on hard pavement or sidewalks. Nor were we designed to clamber around on sharp rocks at high elevations... If you're running on soft ground and you have the foot strength, then there are plenty of arguments in favour of minimalist shoes being better for you than other shoes, because it brings your posture and gait back to what nature intended it to be. If you're hiking in the mountains on uneven rocky terrain, especially while carrying a heavy bag, then you'd be better off wearing a supportive hiking boot...

4. ... My favourite barefoot shoes are Vibram Five-Fingers , I wear either my Mouri's or my KSO's at the beach. Vibram Five-Fingers 'Signa' watersport shoes: The thin sole will be enough to protect your feet from the lava sand...

Затем эти пять результатов добавляются в конец подсказки из листинга 15.6, заменяя значение {resultset} ответом из листинга 15.7. Мы берем окончательную, полностью замененную подсказку и передаем ее в наш LLM, получая следующий результат.

Листинг 15.8. Ответ LLM, связывающий документы с созданием запросов

```
resultset_text = "\n".join(resultset)
resultset_prompt = summarize_search_prompt.replace("{resultset}",
                                                    resultset_text)
generated_relevance_judgments = get_generative_response(resultset_prompt)
display(generated_relevance_judgments)
```

Вывод:

1. What is the definition of a minimalist shoe?
2. What are the characteristics of minimalist shoes?
3. Which shoes are best for walking on rocky beaches?
4. Are minimalist shoes suitable for all terrains?
5. What are some recommended barefoot shoe brands?

Results:

1. [1]
2. [0]
3. [2]
4. [3]
5. [4]

Обратите внимание на нюанс в релевантности между запросами-кандидатами и формулировкой в каждом из их соответствующих результатов. Также обратите внимание, что порядок соответствующих результатов не совпадает с порядком запросов в выходных данных. Контекст документа 0 более релевантен для конкретного запроса *What are the characteristics of minimalist shoes?*, а контекст документа 1 более релевантен для запроса *What is the definition of a minimalist shoe?*

Обратите внимание, что мы просто используем индексную позицию *pandas* для идентификаторов контекста вместо идентификаторов документов. По нашему опыту идентификаторы могут сбивать модель с толку, предоставляя нерелевантную информацию. В зависимости от используемого вами LLM может потребоваться реверсный инжиниринг с некоторым кодом, как мы сделали с нашим существующим *example_contexts* из листинга 15.7. Также обратите внимание, как модель изменила порядок в листинге 15.8 (запросы не в том же порядке, что

и результаты суждений), поэтому нам нужно будет учесть это далее при анализе вывода.

В следующем листинге показано, как извлечь информацию и создать хороший словарь Python для запросов, документов и релевантных результатов.

Листинг 15.9. Извлечение парных суждений из вывода LLM

```
def extract_pairwise_judgments (text, contexts):
    query_pattern = re.compile(r"\d+\.\s+(.*)")
    result_pattern = re.compile(r"\d+\.\s+[(\d+)\]")
    lines = text.split("\n")
    queries = []
    results = []
    for line in lines:
        query_match = query_pattern.match(line)
        result_match = result_pattern.match(line)
        if result_match:
            result_index = int(result_match.group(1))
            results.append(result_index)
        elif query_match:
            query = query_match.group(1)
            queries.append(query)
    output = [{"query": query, "relevant_document": contexts[result]["id"]}
              for query, result in zip(queries, results)]
    return output
```

Регулярные выражения используются для того, чтобы увидеть, какому списку принадлежит та или иная строка. Вы можете поэкспериментировать с более надежными выражениями, если столкнетесь со странным выводом модели.

Далее мы передаем наш вывод из листинга 15.8 в `extract_pairwise_judgments` из листинга 15.9.

Листинг 15.10. Положительные суждения релевантности, полученные из LLM

```
resultset_contexts = example_contexts.to_dict("records")
output = extract_pairwise_judgments (
    generated_relevance_judgments,
    resultset_contexts)
display(output)
```

`example_contexts` из листинга 15.7 содержит результаты поиска по запросу «What are minimalist shoes?»

Ответ:

```
{ "query": "What is the definition of a minimalist shoe?",
  "relevant_document": 18370 }
{ "query": "What are the characteristics of minimalist shoes?",
  "relevant_document": 18376 }
{ "query": "Which shoes are best for walking on rocky beaches?",
  "relevant_document": 18370 }
{ "query": "Are minimalist shoes suitable for all terrains?",
  "relevant_document": 18375 }
{ "query": "What are some recommended barefoot shoe brands?",
  "relevant_document": 13540 }
```

Прогон этого процесса через сотни наборов результатов, взятых из запросов клиентов, даст большой список релевантных пар запрос–документ. Поскольку мы обрабатываем пять результатов за раз, мы также будем генерировать пять новых положительных парных суждений с каждым отдельным запросом к LLM. Затем вы можете использовать эти суждения в качестве синтетических сигналов или в качестве обучающих данных для обучения многих моделей на основе сигналов, рассмотренных в предыдущих главах.

15.2.4. Оценка генеративного вывода

В предыдущих главах мы рассмотрели важность использования списков суждений для измерения качества наших алгоритмов поиска. В главах 10 и 11 мы вручную генерировали списки суждений, а затем делали это автоматически с помощью моделей кликов для обучения и измерения качества моделей LTR. В главе 14 мы генерировали наборы для обучения *silver* и *golden* для измерения качества нашего тонко настроенного LLM для экстрактивного метода ответов на вопросы.

Все это примеры использования поиска, для которых пары запрос–результат в значительной степени детерминированы. Однако измерение качества выходных данных генеративной модели гораздо сложнее. В этом разделе мы рассмотрим некоторые примеры использования генеративной модели, чтобы увидеть, что можно сделать для преодоления этих проблем.

Генеративный вывод может быть субъективным. Многие генеративные модели выдают разные результаты при получении одного и того же запроса, когда вы настраиваете параметр температуры. *Температура* – это значение от 0.0 до 1.0, этот параметр контролирует случайность вывода. Чем ниже температура, тем более предсказуем вывод; чем выше температура, тем более креативен вывод. Мы рекомендуем всегда устанавливать температуру на 0.0 для оценки и создания вашей модели, чтобы вы были еще уверены в ее выводе. Однако даже температура 0.0 может по-прежнему давать разные ответы на один и тот же запрос между запусками в зависимости от модели.

Основная методология оценки генеративного вывода проста: сформулируйте и оцените задачи, которые можно измерить объективно. Однако, поскольку вывод подсказки может быть непредсказуемым, нелегко подобрать набор данных, который можно будет повторно использовать для измерения различных подсказок для вашей конкретной задачи. Поэтому мы обычно полагаемся на установленные метрики для оценки качества выбранной модели, прежде чем вы начнете собственную оценку ее выходных данных.

Различные важные и общие метрики относятся к различным языковым задачам. Каждая метрика обычно имеет таблицу лидеров, которая представляет собой критерий в стиле соревнования, который ранжирует модели по производительности в данных задачах. Вот некоторые общие критерии:

- AI2 Reasoning Challenge (ARC) – набор данных с вопросами и ответами с несколькими вариантами ответов, включающий 7787 вопросов экзамена по естественным наукам для начальной школы;
- HellaSwag – тесты на здравый смысл; например, сложный вопрос на рассуждение дается с несколькими вариантами ответов;
- Massive Multitask Language Understanding (MMLU) – тест для измерения точности многозадачности текстовой модели. Тест охватывает 57 задач, включая математику, историю, информатику, юриспруденцию и многое другое;
- TruthfulQA – измеряет, является ли языковая модель правдивой при генерации ответов на вопросы. Тест включает 817 вопросов, охватывающих 38 категорий. Авторы создали вопросы, на которые некоторые люди ответили бы ложно из-за ложного убеждения или заблуждения.

На рис. 15.3 показан пример теста HellaSwag с вопросами и ответами.

Выберите наиболее подходящую концовку для контекста.

Как ловить стрекоз. Используйте воздушную сеть с длинной ручкой и широким отверстием. Выберите воздушную сеть диаметром 18 дюймов (46 см) или больше.

Выбирайте модель с длинной ручкой.

a) Накиньте 1 кусок ленты на ручку. Поместите шланг или шланг на сетку и надежно завяжите веревку.	b) Дотянитесь ногами до сетки. Двигайте корпус и голову вперед, когда поднимаете ноги.	c) Если возможно, выбирайте темную сетку вместо светлой. Темные сетки стрекозам увидеть сложнее, поэтому им сложнее их избежать.	d) Если она недостаточно прочная для вас, используйте ручную сетку, один конец которой короче другого. Сетка должна иметь отверстия в нижней части.
--	--	--	---

Правильно!

Рис. 15.3. Пример вопроса HellaSwag и несколько вариантов ответов, правильный ответ выделен

Некоторые таблицы лидеров содержат несколько показателей, например таблица Open LLM Leaderboard на рис. 15.4.

Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K	Hub License
abacusai/Smaug-72B-v0.1	80.48	76.02	89.27	77.15	76.67	85.08	78.7	other
ibivibiv/alpaca-dragon-72b-v1	79.3	73.89	88.16	77.4	72.69	86.03	77.63	other
moxeh/MoMo-72B-LoRA-1.8.7-DPO	78.55	70.82	85.96	77.13	74.71	84.06	78.62	mit
cloudyu/TomGxc_FusionNet_34Bx2_MoE_v0.1_DPO_f16	77.91	74.06	86.74	76.65	72.24	83.35	74.45	other
HanNayeonLee/LHK_DPO_v1	77.62	74.74	89.3	64.9	79.89	88.32	68.54	mit
cloudyu/TomGxc_FusionNet_34Bx2_MoE_v0.1_full_linear_DPO	77.52	74.06	86.67	76.69	71.32	83.43	72.93	other
zhengx/MixTAO-7Bx2-MoE-v0.1	77.5	73.81	89.22	64.92	78.57	87.37	71.11	apache-2.0
yunconglong/Truthful_DPO_TomGxc_FusionNet_7Bx2_MoE_13B	77.44	74.91	89.3	64.67	78.02	88.24	69.52	mit
JaeyeonKang/CCK_Asuza_v1	77.43	73.89	89.07	75.44	71.75	86.35	68.08	cc-by-nc-4.0
fbiglit/UNA-SimpleSmaug-34b-v1beta	77.41	74.57	86.74	76.68	70.17	83.82	72.48	apache-2.0
TomGxc/FusionNet_34Bx2_MoE_v0.1	77.38	73.72	86.46	76.72	71.01	83.35	73.01	mit
midtissex/Tess-72B-v1.5b	77.3	71.25	85.53	76.63	71.99	81.45	76.95	other

Рис. 15.4. Пространство таблицы лидеров HuggingFaceH4 Open LLM (взято в марте 2024 года)

Вы должны использовать эти метрики, чтобы решить, какую модель использовать для вашей конкретной области и задачи. Например, если ваша задача – абстрактный ответ на вопрос, рассмотрите возможность использования модели с лучшей метрикой TruthfulQA.

ПРЕДУПРЕЖДЕНИЕ. Модели лицензированы, поэтому убедитесь, что вы используете модель с лицензией, которая соответствует вашему варианту использования. Некоторые модели имеют лицензии, похожие на программное обеспечение с открытым исходным кодом (например, Apache 2.0 или MIT). Но будьте осторожны, поскольку многие модели имеют коммерческие ограничения (например, модели LLaMA и производные или модели, обученные на выходных данных из ограничительных моделей, таких как GPT из OpenAI).

После того как вы выбрали свою модель, нужно предпринять шаги для ее оценки по вашим подсказкам. Будьте строги и отслеживайте ответы на ваши подсказки и то, как они проходят. На рынке появляются некоторые специализированные инструменты, которые делают это для вас, включая подходы к автоматической оптимизации подсказок, но может быть достаточно простой электронной таблицы. В следующем разделе мы построим собственную метрику для использования в новой задаче.

15.2.5. Построение вашей собственной метрики

Давайте рассмотрим, как можно объективно интерпретировать генеративную задачу, позволяя вам создать набор данных и использовать такую метрику, как точность или отзыв (оба были представлены в разделе 5.4.5), для оценки точности модели. Мы будем использовать подсказку с неструктурированными данными для извлечения структурированной информации. Поскольку наш результат будет отформатирован предсказуемо, мы затем сможем измерить точность ответа на основе результата, маркированного человеком.

В нашем примере мы проверим точность распознавания именованных сущностей генеративной модели. В частности, мы измерим, правильно ли генеративный вывод из LLM маркирует сущности типа «человек», «организация» или «локация». Мы будем использовать следующий фрагмент текста из новостной статьи:

Walter Silverman of Brighton owned one of the most successful local Carvel franchises, at East Ridge Road and Hudson Avenue in Irondequoit. He started working for Carvel in 1952. This is how it appeared in the late 1970s/early 1980s.
[Alan Morrell, Democrat and Chronicle, May 29, 2023]

Если мы вручную пометим сущности в этом фрагменте тегами <per> (person), <org> (organization) и <loc> (location), то придем к помеченной версии в следующем листинге.

Листинг 15.11. Статья, маркированная метками сущностей

<per>Walter Silverman</per> of <loc>Brighton</loc> owned one of the most successful local <org>Carvel</org> franchises, at <loc>East Ridge Road</loc> and <loc>Hudson Avenue</loc> in <loc>Irondequoit</loc>. He started working for <org>Carvel</org> in 1952. This is how it appeared in the late 1970s/early 1980s.

Это формат, который должна уметь создавать генеративная модель, когда берет текст и генерирует размеченную версию в соответствии с указаниями подсказки. Но этот ответ является лишь полуструктурированным с помощью специальной разметки. Нам нужно обработать его дальше, и мы можем написать немного кода Python для извлечения сущностей в подходящий формат JSON.

Листинг 15.12. Извлечение сущностей из генеративного вывода

```
def extract_entities(text):
    entities = []
    pattern = r"<(per|loc|org)>(.*?)</(per|loc|org)>"
    matches = re.finditer(pattern, text)
    for match in matches:
        entity = {"label": match.group(1).upper(),
                  "offset": [match.start(), match.end() - 1],
                  "text": match.group(2)}
        entities.append(entity)
    return entities

entities = extract_entities(news_article_labelled)
display(entities)
```

Вывод:

```
[{"label": "PER", "offset": [0, 26], "text": "Walter Silverman"},
 {"label": "LOC", "offset": [31, 49], "text": "Brighton"},
 {"label": "ORG", "offset": [90, 106], "text": "Carvel"},
 {"label": "LOC", "offset": [123, 148], "text": "East Ridge Road"},
 {"label": "LOC", "offset": [154, 177], "text": "Hudson Avenue"},
 {"label": "LOC", "offset": [182, 203], "text": "Irondequoit"},
 {"label": "ORG", "offset": [229, 245], "text": "Carvel"}]
```

Теперь, когда сущности представлены в структурированной форме (как JSON), мы можем использовать список для расчета нашей метрики. Мы можем вручную маркировать множество отрывков или использовать методы из разделов 14.2–14.3 для автоматизации маркировки, используя модель для построения набора silver, а затем корректируя выходные данные для получения набора golden.

Когда у вас есть набор golden, вы можете попробовать подсказку, подобную следующей.

Листинг 15.13. Маркированные сущности

```
For a given passage, please identify and mark the following entities:
people with the tag '<per>', locations with the tag '<loc>', and
organizations with the tag '<org>'. Please repeat the passage below
with the appropriate markup.
### {text}
```

В предыдущей подсказке {text} будет заменен абзацем, для которого вы хотите идентифицировать сущности. Когда модель генерирует ответ, его можно передать непосредственно в функцию `extract_entities` в листинге 15.12.

Выходные данные извлеченных сущностей затем можно сравнить с набором `golden`, чтобы получить количество истинно положительных результатов, TP (правильно идентифицированные сущности), ложноположительных результатов, FP (неправильно идентифицированный текст, который не должен быть идентифицирован), и ложноотрицательных результатов, FN (сущности, которые должны были быть идентифицированы, но не были).

Из этих чисел вы можете вычислить следующее:

- $\text{Precision} = (\text{TP} / (\text{TP} + \text{FP}))$,
- $\text{Recall} = (\text{TP} / (\text{TP} + \text{FN}))$,
- $\text{F1} = (2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}))$.

Многие из показателей, перечисленных в таблицах лидеров, используют оценки F1.

Какие генеративные задачи у вас есть? Будьте креативны и подумайте о том, как вы можете сформировать свою задачу, чтобы она была объективно измеримой. Например, если вы создаете сводки результатов поиска, можете выполнить что-то похожее на задачу распознавания именованных сущностей – проверить, показывают ли сводки важные фрагменты текста для заданного набора результатов поиска или показывают ли ссылки на номера результатов с правильно указанными заголовками.

15.2.6. Оптимизация алгоритмических подсказок

В разделе 15.2 мы рассмотрели использование подсказок LLM для создания синтетических данных и для обобщения и цитирования результатов поиска для RAG. Мы также рассмотрели метрики для количественной оценки качества сгенерированных данных. Мы обсуждали это в контексте измерения качества модели, но есть важный момент, который мы упустили из виду: качество подсказки.

Если вы помните, у нас есть три разных способа улучшить вывод LLM: улучшить модель во время обучения (предобучение), тонко настроить модель или улучшить подсказку. Предварительное обучение и тонкая настройка – это прямые программные оптимизации модели нейронной сети, но до сих пор мы рассматривали создание и оптимизацию подсказок (известные как *техника подсказок*) как ручной процесс, требующий вмешательства человека.

На самом деле подсказку можно также тонко настроить программно, как и модель. Такие библиотеки, как DSPy (Declarative Self-improving Language Programs, Pythonically), предоставляют основу для программирования использования языковой модели, а не для ручного ее подсказывания. Функционально это достигается путем определения желаемого формата вывода из языковой модели (вместе с данными обучения, показывающими хорошие ответы) и предоставления библиотеке возможности оптимизировать формулировку подсказки для наилучшего достижения этого вывода.

DSPy выполняет эту программную оптимизацию подсказки, используя четыре ключевых компонента:

- **сигнатуры** – это простые строки, объясняющие цель процесса обучения, а также ожидаемые входные и выходные данные. Примеры:
 - `question` → `answer` (ответ на вопрос),
 - `document` → `summary` (обобщение документа),
 - `context, question` → `answer` (RAG);
- **модули** – они берут сигнатуру и объединяют ее с методом подсказок, настроенным LLM и набором параметров для генерации подсказки для достижения желаемого результата;
- **метрики** – это меры того, насколько хорошо вывод, сгенерированный модулем, соответствует желаемому результату;
- **оптимизаторы** – они используются для обучения модели путем итеративного тестирования вариаций подсказок и параметров для оптимизации для указанной метрики.

Таким образом, программирование использования языковой модели дает четыре ключевых преимущества:

- оно позволяет автоматически оптимизировать подсказку, что может быть трудоемким и подверженным ошибкам процессом, если выполнять его вручную;
- позволяет оптимизировать подсказку для быстрого тестирования известных шаблонов (и новых методов или данных с течением времени) и лучших практик для подсказок, а также связать несколько этапов подсказок в более сложный конвейер;
- позволяет вам легко переключать LLM в любое время и «перекомпилировать» программу DSPy для повторной оптимизации подсказки для новой модели;
- позволяет вам связать несколько модулей, моделей и рабочих процессов в более сложные конвейеры и оптимизировать подсказки для каждого этапа конвейера и конвейера в целом.

Эти преимущества делают ваше приложение более надежным и простым в обслуживании по сравнению с ручной настройкой подсказок, и они гарантируют, что ваша подсказка всегда будет точно настроена и оптимизирована для вашей текущей среды и целей. После того как вы закодировали настроенный конвейер DSPy, вы просто запускаете оптимизатор для его *компиляции*, после чего все оптимальные параметры изучаются для каждого модуля. Хотите затем попро-

бовать новую модель, метод или набор обучающих данных? Просто перекомпилируйте свою программу DSPy, и вы готовы к работе, а ваши подсказки будут автоматически оптимизированы для получения наилучших результатов. Вы можете найти множество шаблонов «начала работы» для реализации RAG, ответов на вопросы, резюмирования и любого количества других задач на основе подсказок LLM в документации DSPy по адресу <https://github.com/stanfordnlp/dspy>. Если вы создаете нетривиальное приложение на основе LLM, мы настоятельно рекомендуем вам выбрать такой программный подход к оптимизации подсказок.

15.3. Мультимодальный поиск

Давайте теперь рассмотрим одну из самых мощных возможностей, предоставляемых эмбедингами на основе базовых моделей: мультимодальный поиск. Мультимодальные поисковые системы позволяют вам искать один тип контента, используя другой тип контента (также называемый кросс-модальным поиском), или искать несколько типов контента вместе. Например, вы можете искать изображение, используя текст, изображение или гибридный запрос, объединяющий текст и изображение. Мы реализуем каждый из этих вариантов использования в этом разделе.

Мультимодальный поиск стал возможным благодаря тому, что векторные эмбединги могут быть сгенерированы для текста, изображений, аудио, видео и другого контента, которые могут быть отображены в перекрывающихся векторных пространствах. Это позволяет нам искать любой тип контента по любому другому типу контента без необходимости какой-либо специальной индексации или преобразования.

Мультимодальные поисковые системы имеют потенциал для революционного изменения способа поиска информации, устраняя барьеры, которые ранее мешали нам использовать весь доступный контекст для наилучшего понимания входящих запросов и ранжирования результатов.

15.3.1. Общие режимы мультимодального поиска

Хотя мультимодальный поиск подразумевает, что для поиска доступны многие различные типы данных, в этом разделе мы кратко обсудим наиболее распространенные в настоящее время модальности данных: естественный язык, изображения, аудио и видео.

Поиск на естественном языке

В главах 13 и 14 мы реализовали семантический поиск и вопросно-ответный поиск с использованием плотных векторов. Оба метода являются примерами поиска на естественном языке. Простой запрос, например *shirt without stripes* («рубашка без полос»), запутает любую платформу электронной коммерции, построенную на основе инвер-

тированного индекса, если только специальная логика не будет добавлена вручную или интегрирована в ориентированный на поиск граф знаний с использованием *семантических функций*, подобных тем, что мы реализовали в главе 7.

Современные LLM становятся все более искушенными в обработке этих запросов и даже способны интерпретировать инструкции и синтезировать информацию из разных источников для генерации новых ответов. Многие компании, создающие традиционные базы данных и хранилища данных NoSQL, активно интегрируют поддержку плотных векторов в качестве первоклассных типов данных, чтобы воспользоваться всеми этими основными достижениями.

Хотя маловероятно, что эти подходы на основе плотных векторов полностью заменят традиционные инвертированные индексы, графы знаний, бустинг сигналов, обучение ранжированию, кликовые модели и другие методы поиска, мы увидим появление новых гибридных подходов, которые объединят лучшее из каждого из этих методов для продолжения оптимизации контента и понимания запросов.

Поиск изображений

Как вы видели в главе 2, изображения являются еще одной формой неструктурированных данных, наряду с текстом.

Традиционно, если вы хотите найти изображения в инвертированном индексе, вы будете искать по текстовым описаниям изображения или меткам, примененным к изображению. Однако с помощью плотного векторного поиска поиск изображений по визуальному или понятийному сходству может быть реализован почти так же легко, как и текстовый поиск. Беря входные изображения, кодируя их в векторы эмбединга с помощью визуального трансформера¹ и индексируя эти векторы, становится возможным принять другое изображение в качестве запроса, кодировать его в эмбединг, а затем выполнить поиск с помощью этого вектора, чтобы найти визуально похожие изображения.

Кроме того, обучая модель с изображениями вместе с текстовыми описаниями этих изображений, становится возможным выполнять мультимодальный поиск изображений либо путем ввода изображения, либо по тексту. Вместо того чтобы иметь подпись и бесполезное для поиска изображение, теперь можно использовать сверточные нейронные сети (CNN) или визуальные трансформеры для кодирования изображения в плотное векторное пространство, которое сосуществует в том же векторном пространстве, что

¹ Визуальные трансформеры, англ. *Vision Transformers, ViTs*, – это класс моделей глубокого обучения, которые используют архитектуру трансформеров для задач обработки изображений. На выходе трансформера получается последовательность векторов, каждый из которых представляет фрагмент изображения. Последний вектор используется для задач классификации. Его пропускают через линейный слой, за которым следует функция softmax для получения вероятностей классов. – *Прим. ред.*

и текст. С представлением изображения и текстовым представлением в том же векторном пространстве вы можете искать более описательную версию изображения, чем предоставляет подпись, и при этом по-прежнему сопоставлять, используя текст для описания особенностей на изображении.

Учитывая эту возможность кодировать изображения и текст в одно и то же плотное векторное пространство, также возможно обратить поиск и использовать изображение в качестве запроса для сопоставления любых текстовых документов, которые включают похожий язык с тем, что происходит на изображении, или объединить изображение и текст в гибридный запрос, чтобы найти другие изображения, наилучшим образом соответствующие комбинации модальностей запроса текста и изображения. Мы рассмотрим примеры некоторых из этих методов в разделе 15.3.2.

Поиск аудио

Поиск аудио или по аудио исторически был в основном проблемой преобразования текста в речь. Аудиофайлы должны быть преобразованы в текст и проиндексированы, а аудиозапросы преобразованы в текст и проанализированы по текстовому индексу. Транскрибирование голоса является очень сложной проблемой. Голосовые помощники от Google, Apple, Amazon и Microsoft обычно очень хорошо справляются с короткими запросами, но этот уровень точности исторически был трудно достижим для тех, кто хотел интегрировать решения с открытым исходным кодом. Однако недавние прорывы в сочетании с моделями распознавания речи и языка, которые могут исправлять фонетические недоразумения, выводят на рынок более совершенные технологии.

Важно помнить, что в аудио могут присутствовать многие другие важные звуки, помимо просто произнесенных слов. Если кто-то ищет *loud train, rushing stream* или *Irish accent*, все эти качества он надеется найти в любых возвращенных результатах аудио. Модели мультимодальных трансформеров¹, обученные сопоставлению текста и звука с перекрывающимися векторными пространствами, теперь делают это возможным.

Поиск видео

Видео – это не что иное, как комбинация последовательных изображений (кадров), наложенных и сохраненных в последовательности со звуком. Если звук и изображения можно сопоставить с перекрывающимся векторным пространством с письменным текстом, это означает, что путем индексации каждого кадра видео (или, может

¹ Мультимодальный трансформер, англ. *Multimodal Transformer*, – это модель, которая расширяет архитектуру трансформера для включения других модальностей, таких как изображения и аудио. Основная идея заключается в том, чтобы закодировать разные модальности отдельно, а затем объединить их. – Прим. ред.

быть, нескольких кадров в секунду, в зависимости от необходимой детализации) можно создать поисковую систему видео, которая позволяет выполнять поиск по текстовому описанию для любой сцены в видео, поиск по аудиоклипу или поиск по изображению, чтобы найти наиболее похожее видео. Модели глубокого обучения, создаваемые в областях компьютерного зрения, обработки естественного языка и обработки звука, постепенно сходятся, и пока эти различные типы медиа могут быть представлены в перекрывающихся векторных пространствах, поисковые системы теперь могут осуществлять их поиск.

15.3.2. Реализация мультимодального поиска

В этом разделе мы реализуем одну из самых популярных форм мультимодального поиска: поиск текст–изображение. Мы сделаем это, используя эмбединги, которые были совместно обучены на парах данных изображение–текст из интернета. Это приводит к тому, что изученные латентные признаки текста находятся в том же векторном пространстве, что и изученные латентные признаки изображения. Это также означает, что в дополнение к выполнению поиска текст–изображение мы можем использовать ту же модель эмбединга для выполнения визуального поиска изображение–изображение на основе пикселей в любом входящем изображении. Мы также покажем пример объединения модальностей в гибридном запросе, делая поиск результатов, которые наилучшим образом соответствуют как текстовому запросу, так и изображению *одновременно*.

Мы будем использовать модель CLIP, которая является мультимодальной моделью, способной понимать изображения и текст в одном и том же векторном пространстве. CLIP – это модель на основе трансформера, разработанная OpenAI, которая была обучена на большом наборе данных изображений и связанных с ними текстовых подписей. Модель была обучена предсказывать, какая подпись подходит к какому изображению, и наоборот.

Это означает, что модель научилась сопоставлять изображения и текст в одном и том же векторном пространстве, чтобы похожие изображения и связанные текстовые эмбединги располагались близко друг к другу.

Мы вернемся к набору данных Movie Database (TMDB), который использовали в главе 10 для реализации наших примеров мультимодального поиска. Однако в этом случае вместо поиска по тексту фильмов мы будем искать по изображениям (кадрам) из фильмов.

В следующем листинге мы определяем функциональность для вычисления нормированных вставок и создания коллекции фильмов. Коллекция состоит из вставок кадров фильма и метаданных фильма, включая заголовки, URL-адреса источников изображений и URL-адрес страницы фильма в TMDB.

Листинг 15.14. Индексирование кешированных эмбедингов фильмов

```
def normalize_embedding(embedding):
    return numpy.divide(embedding,
        numpy.linalg.norm(embedding,axis=0)).tolist()

def read(cache_name):
    cache_file_name = f"data/tmdb/{cache_name}.pickle"
    with open(cache_file_name, "rb") as fd:
        return pickle.load(fd)

    Мы предварительно сгенерировали и кешировали эмбединги фильмов, поэтому вам не придется загружать и обрабатывать все кадры.

def tmdb_with_embeddings_dataframe():
    movies = read("movies_with_image_embeddings")
    embeddings = movies["image_embeddings"]
    normalized_embeddings = [normalize_embedding(e)
        for e in embeddings]
    movies_dataframe = spark.createDataFrame(
        zip(movies["movie_ids"], movies["titles"],
            movies["image_ids"], normalized_embeddings),
        schema=["movie_id", "title", "image_id", "image_embedding"])
    return movies_dataframe

embeddings_dataframe = tmdb_with_embeddings_dataframe()
embeddings_collection = engine.create_collection(
    "tmdb_with_embeddings")
embeddings_collection.write(embeddings_dataframe)
```

Нормирует эмбединги фильмов во время индексирования, чтобы мы могли использовать более эффективный расчет скалярного произведения (по сравнению с косинусом) во время запроса.

Создает коллекцию фильмов, содержащую эмбединги их кадров.

Поскольку модель CLIP является мультимодальной моделью, совместно обученной на тексте и изображениях, эмбединги, которые мы генерируем из текста или изображений, можно использовать для поиска тех же эмбедингов изображений.

Теперь, когда индекс создан с использованием нормированных эмбедингов, все, что осталось сделать, – это выполнить векторный поиск по коллекции. В следующем листинге показаны основные функции, необходимые для поиска изображений с использованием текстовых запросов, запросов изображений или гибридных текстовых и графических запросов.

Листинг 15.15. Мультимодальный векторный поиск текста и изображения с использованием CLIP

```
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)
```

Загружает предварительно обученную модель CLIP и препроцессор изображений.

Выполняет векторный поиск для эмбеддингов запроса.

```
def movie_search(query_embedding, limit=8):
    collection = engine.get_collection("tmdb_with_embeddings")
    request = {"query_vector": query_embedding,
               "query_field": "image_embedding",
               "return_fields": ["movie_id", "title",
                                "image_id", "score"],
               "limit": limit,
               "quantization_size": "FLOAT32"}
    return collection.search(**request)
```

Создает поисковый запрос с эмбеддингом запроса для поиска по индексированным эмбеддингам изображений.

Тип данных, используемый для каждого значения функции эмбеддинга, в данном случае 32-битное число с плавающей точкой.

```
def encode_text(text, normalize=True):
    text = clip.tokenize([text]).to(device)
    text_features = model.encode_text(text)
    embedding = text_features.tolist()[0]
    if normalize:
        embedding = normalize_embedding(embedding)
    return embedding
```

Вычисляет нормированные эмбеддинги для текста.

```
def encode_image(image_file, normalize=True):
    image = load_image(image_file)
    inputs = preprocess(image).unsqueeze(0).to(device)
    embedding = model.encode_image(inputs).tolist()[0]
    if normalize:
        embedding = normalize_embedding(embedding)
    return embedding
```

Вычисляет нормированные эмбеддинги для изображения.

```
def encode_text_and_image(text_query, image_file):
    text_embedding = encode_text(text_query, False)
    image_embedding = encode_image(image_file, False)
    return numpy.average((normalize_embedding(
        [text_embedding, image_embedding])), axis=0).tolist()
```

Вычисляет и объединяет нормированные эмбеддинги для изображения и текста.

Усредняет векторы текста и изображения для создания мультимодального запроса.

Функция `movie_search` следует процессу, похожему на тот, который мы использовали в главе 13: берем вектор запроса и выполняем поиск вектора по коллекции с эмбеддингами. Наши функции `encode_text` и `encode_image` вычисляют нормированные эмбеддинги на основе текста или изображения. Функция `encode_text_and_image` является гибридом этих двух, где мы генерируем эмбеддинги как из текста, так и из изображений, нормируем их по единицам и объединяем их вместе путем усреднения.

С основными вычислениями мультимодального эмбеддинга теперь мы можем реализовать простой интерфейс поиска для отображения лучших результатов для входящего запроса текста, изображения или текста и изображения вместе.

Листинг 15.16. Мультимодальный векторный поиск по эмбедингам текста и изображения

```
def search_and_display(text_query="", image_query=None):
    if image_query:
        if text_query:
            query_embedding = encode_text_and_image(text_query, image_query)
        else:
            query_embedding = encode_image(image_query)
    else:
        query_embedding = encode_text(text_query)
    display_results(movie_search(query_embedding), show_fields=False)
```

Функция `search_and_display` принимает либо текстовый запрос, либо запрос изображения, либо оба, а затем извлекает эмбединги и выполняет поиск. Затем функция отображает лучшие результаты в простой сетке. На рис. 15.5 показан пример вывода для запроса `singing in the rain`:

```
search_and_display(text_query="singing in the rain")
```

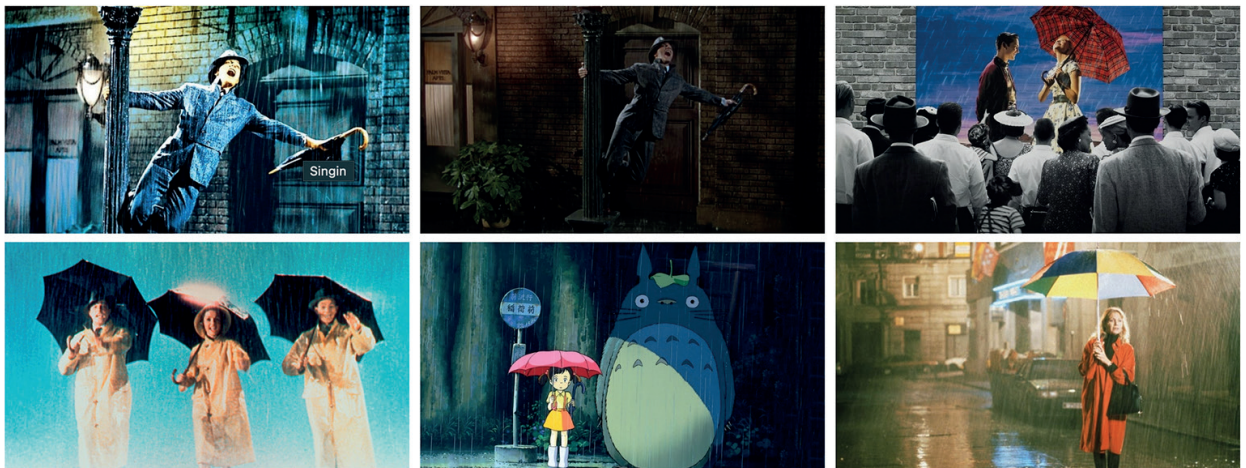


Рис. 15.5. Текстовый запрос для `singing in the rain`

Три из первых четырех изображений взяты из фильма «Поющие под дождем», все изображения показывают кого-то под дождем или с зонтиком, а многие из них либо из мюзиклов, либо показывают людей, которые активно поют. Эти результаты демонстрируют мощь поиска изображений с использованием эмбедингов, которые были обучены как на тексте, так и на изображениях: теперь у нас есть возможность искать изображения, пиксели которых содержат значение, выраженное словами!

Чтобы продемонстрировать некоторые нюансы сопоставления текста со значением в изображениях, давайте запустим два варианта одного и того же запроса: `superhero flying` и `superheroes flying`. В традиционном поиске по ключевым словам мы обычно отбрасываем множественное число, но в контексте поиска по эмбедингам, и особенно в контексте мультимодального поиска, давайте посмотрим, как это небольшое различие влияет на результаты. Как вы можете видеть на

рис. 15.6, наши результаты поиска изменились с изображений одного летящего супергероя на изображения в основном групп супергероев, содержащих схожие действия.

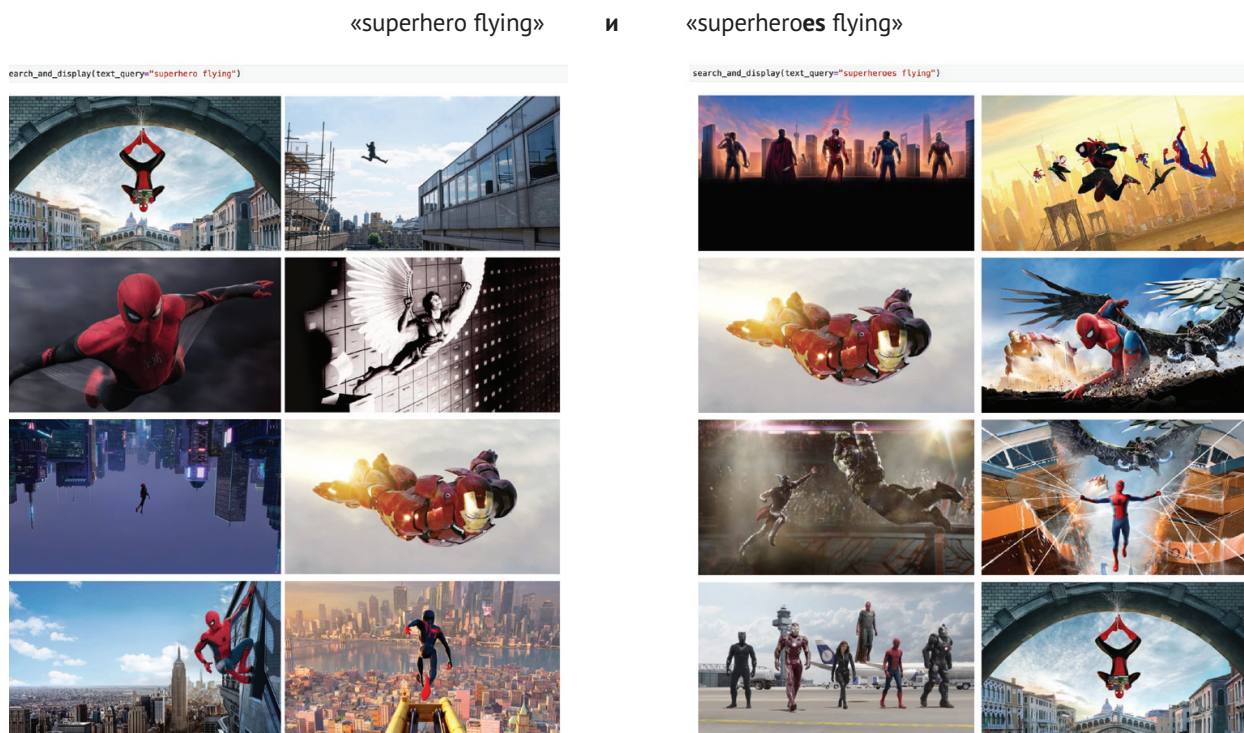


Рис. 15.6. Тонкая разница между запросами superhero flying и superheroes flying

Этот кросс-модальный (текст–изображение) поиск впечатляет, но давайте также протестируем поиск изображение–изображение, чтобы продемонстрировать, насколько мультимодальные эмбединги можно использовать повторно. На рис. 15.7 показаны результаты поиска изображений с использованием изображения DeLorean (автомобиль, который, как известно, превращен в машину времени в фильме «Назад в будущее»).

```
search_and_display(image_query="chapters/ch15/delorean-query.jpg")
```

Как и ожидалось, большинство автомобилей на самом деле являются знаменитыми DeLorean из «Назад в будущее». У других автомобилей схожие эстетические черты: похожие формы, похожие открывающиеся вверх двери. Мы не только хорошо сопоставили черты автомобиля, но и многие результаты также отражают и свечение из запроса изображения.

Чтобы продвинуть наш поиск по DeLorean на шаг дальше, давайте попробуем мультимодальный поиск, объединив наши последние два примера запросов. Давайте выполним мультимодальный запрос как для предыдущего изображения DeLorean (модальность изображения), так и для текстового запроса superhero (модальность текста). Вывод этого запроса показан на рис. 15.8.

```
search_and_display(text_query=»superhero»,
                    image_query=»chapters/ch15/delorean-query.jpg»)
```


Запрос изображения:



Результаты запроса изображения:

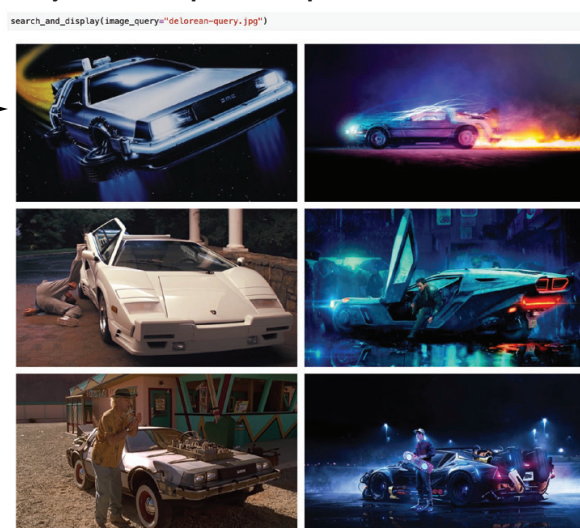


Рис. 15.7. Поиск по изображению автомобиля, похожего на DeLorean

Запрос изображения:



+

Текстовый запрос:
"superhero"

Результаты мультимодального запроса:

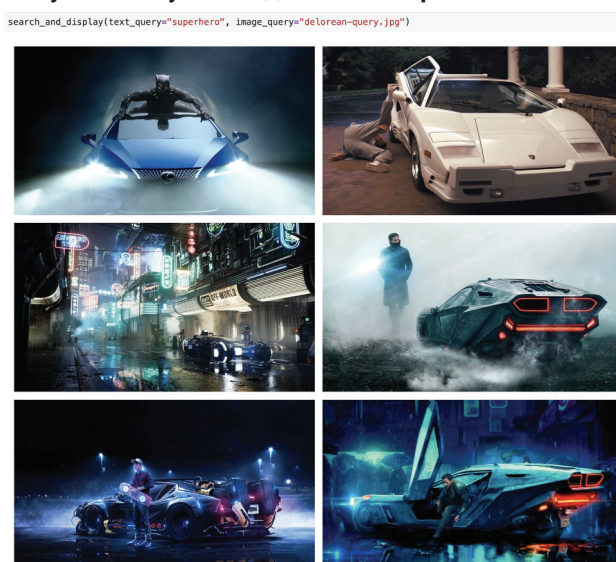


Рис. 15.8. Мультимодальный поиск для запроса изображения и текста

Интересно! На первом изображении показан супергерой (Черная пантера) на спортивной машине, фары машины светятся, как на оригинальной картинке. Большинство результатов теперь показывают главных героев фильмов, причем каждый из результатов содержит спортивные автомобили, а большинство изображений содержат световые эффекты, взятые из запроса изображения. Это отличный пример того, как мультимодальные эмбединги могут использоваться для вывода нюансов намерения и для обеспечения возможности новых способов запросов в различных типах наборов данных.

Последствия мультимодального поиска по тексту и изображениям огромны. В поиске электронной коммерции кто-то может загрузить изображение нужной ему детали и немедленно найти ее, не зная названия, или загрузить изображение предмета одежды или мебели, ко-

которые ему нравятся, и найти похожие стили. Кто-то может описать медицинскую проблему и немедленно найти изображения, отражающие симптомы или соответствующие болезни органы. Из главы 9 вы, возможно, помните нашу работу по изучению латентных признаков предметов и пользователей из сигналов поведения пользователя. Это поведение является еще одной модальностью, подобно изображениям, которую можно изучить и перекрестно обучить с текстом и другими модальностями для повышения интеллекта вашей поисковой системы. Как мы обсудим далее в разделе 15.5, многие дополнительные модальности данных, подобные этой, продолжают становиться общедоступными для поиска в будущем, помогая сделать поисковые системы еще более мощными в понимании и поиске релевантного контента и ответов.

15.4. Другие появляющиеся парадигмы поиска на основе ИИ

В этой книге вы изучили методы в большинстве ключевых категорий поиска на основе ИИ, таких как обработка сигналов и краудсорсинговая релевантность, графы знаний, персонализированный поиск, обучение ранжированию и семантический поиск с использованием эмбедингов на основе трансформеров.

Эмбединг базовых моделей и возможность представления языка в виде плотных векторов в последние годы произвели революцию в том, как мы думаем о поисковых системах и строим их. Если раньше поиск был в основном текстовым, то теперь можно искать по всему, что можно закодировать в плотные векторы. Это включает поиск по изображениям, аудио, видео, понятиям или любой комбинации этих или других модальностей.

Кроме того, способ взаимодействия пользователей с поиском продолжает развиваться новыми и оригинальными способами. Это включает в себя возможность задавать вопросы, получать обобщенные ответы, которые представляют собой комбинацию нескольких результатов поиска, генерировать новый контент, объясняющий результаты поиска, и генерировать изображения, текст, код, инструкции или другой контент. Интерфейсы чат-ботов в сочетании с отслеживанием контекста во время разговоров позволили итеративно совершенствовать задачи поиска, где пользователи могут предоставлять обратную связь в реальном времени по ответам и позволяют ИИ выступать в качестве агента для поиска, синтеза и уточнения релевантности результатов поиска на основе живой обратной связи с пользователями.

В этом заключительном разделе нашего путешествия по поиску с использованием ИИ мы коснемся нескольких тенденций, которые наблюдаем, и того, как они, вероятно, сформируют будущее поиска.

15.4.1. Разговорный и контекстный поиск

Мы потратили значительное время на изучение контекстного поиска – понимание контекста контента, контекста домена и контекста

пользователя при интерпретации запросов. По мере того как поисковые системы становятся более разговорными, личный контекст начинает приобретать еще больший приоритет. Например,

User: "Please, take me home."
Phone: "I don't know where you live."

(Пользователь: «Пожалуйста, отведи меня домой».)
Телефон: «Я не знаю, где ты живешь».)

Несмотря на то что кто-то имеет свой адрес в закладках под названием «Дом» и его телефон ежедневно отслеживает его локацию (включая место, где он спит ночью), некоторые часто используемые цифровые помощники все еще не могут автоматически забирать этот контекст, как на рис. 15.9, и вместо этого требуют, чтобы ваш домашний адрес был явно указан в настройках. Эти недостатки, вероятно, исчезнут в ближайшие годы, поскольку персональные помощники создают гораздо более точные модели персонализации из всех доступных контекстов.

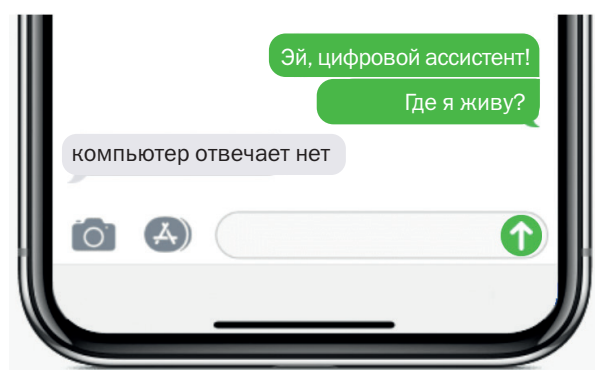


Рис. 15.9. Цифровым помощникам нужно будет извлечь множество различных источников данных для создания надежных моделей персонализации

В дополнение к этим контекстам персонализации есть дополнительный контекст, который становится все более важным: разговорный контекст. Вот пример чат-бота (виртуального помощника), которому не хватает осведомленности о текущем разговорном контексте:

User: "Take me to the plumbing store."

Phone: "Here is a list of stores that I found. [shares list]"

User: "Go to the one in East Rochester."

Phone: [Provides directions to the geographical center of East Rochester, not the plumbing store it suggested in the last exchange]

(«Пользователь: «Отведи меня в магазин сантехники».

Телефон: «Вот список магазинов, которые я нашел. [дает список]»

Пользователь: «Идем в тот, что в Ист-Рочестере».

Телефон: [Указывает направление к географическому центру Ист-Рочестера, а не к магазину сантехники, который был предложен в последнем сообщении])

Хорошие чат-боты интегрируют сильный поисковый индекс, чтобы усилить свою способность отвечать на вопросы и возвращать информацию, но у многих по-прежнему отсутствует краткосрочная

и долгосрочная память о предыдущем разговорном контексте. Они могут искать каждый запрос независимо, не принимая во внимание предыдущие запросы и ответы. В зависимости от приложения и типа запроса, некоторый предыдущий разговорный контекст имеет смысл помнить бесконечно как часть модели персонализации, например домашний адрес человека, имена членов семьи или часто посещаемые места (врач, стоматолог, школа и т. д.). Во многих случаях, однако, достаточно краткосрочной памяти только во время текущего разговора. Поскольку поисковые опыты становятся более разговорными, краткосрочная и долгосрочная память о разговорном контексте становится более важной.

Фундаментальные модели, обученные для чат-контекстов, такие как ChatGPT от OpenAI, привели к радикальному улучшению разговорных возможностей чат-ботов. Хотя они являются предварительно обученными моделями, которые не обновляются в режиме реального времени, в их подсказки можно вводить дополнительный контекст, что делает вполне возможным захват и добавление персонализированных контекстов в будущие подсказки для улучшения их способности обслуживать каждого пользователя.

15.4.2. Поиск на основе агентов

Точно так же, как в главе 7 мы обсуждали важность конвейеров для лучшей интерпретации и представления смысла пользовательских запросов, конвейеры являются неотъемлемым компонентом для многошагового решения проблем, которое возникает с чат-ботами и взаимодействиями базовой модели.

При поиске на основе агентов поисковая система или другой интерфейс ИИ получает подсказку и затем может генерировать новые подсказки или задачи, объединяя их вместе для достижения желаемого результата. Решение запроса пользователя может потребовать нескольких шагов, таких как создание новых задач («теперь найдите лучшие веб-сайты по этой теме» или «объясните, был ли ваш последний ответ правильным»), рассуждения о данных («теперь объедините списки, полученные с каждого веб-сайта» или «суммируйте результаты») и другие функциональные шаги («теперь верните результаты поиска»).

Значительная часть веб-трафика в будущем будет исходить от агентов на основе ИИ, ищущих информацию для использования в решении своих проблем. Степень, в которой поисковые системы выступают в качестве уровня обслуживания данных для этих агентов, поскольку поисковые системы обычно уже имеют кешированную копию большей части сети, еще предстоит выяснить, но поисковые системы являются естественными точками запуска для этих агентов на основе ИИ. Крупные поисковые системы, такие как Bing, а также несколько стартапов, уже развернули многошаговый поиск, включающий определенный уровень итеративного следования задачам для составления более исследованных ответов. Это, вероятно, будет растущей тенденцией в течение некоторого времени в будущем.

15.5. Гибридный поиск

В главах 2 и 3 мы представили две различные парадигмы поиска: лексический поиск на основе ключевых слов (основанный на разреженных векторных представлениях, таких как инвертированный индекс, и обычно ранжируемый с использованием BM25) и поиск по плотным векторам (основанный на плотных векторных представлениях и обычно ранжируемый с использованием косинуса, скалярного произведения или аналогичного расчета подобия на основе векторов). В этом разделе мы покажем, как объединить результаты в этих различных парадигмах.

Мы в основном рассматривали лексический и векторный поиск как ортогональные подходы к поиску, но на практике вы часто получите наилучшие результаты, используя оба подхода как часть гибридного поиска. *Гибридный поиск* – это процесс объединения результатов из нескольких поисковых парадигм для предоставления наиболее релевантных результатов. Гибридный поиск обычно включает в себя объединение результатов лексического и векторного поиска, хотя этот термин не ограничивается только этими двумя подходами.

Существует несколько способов реализации гибридного поиска. Некоторые системы напрямую поддерживают объединение синтаксиса лексического запроса и синтаксиса векторного поиска в рамках одного запроса, что позволяет вам эффективно заменять определенные ключевые слова или фильтры в вашем запросе векторами и объединять оценки сходства векторов, оценки BM25 и оценки функций произвольно сложными способами. Некоторые системы не поддерживают запуск векторного и лексического поиска в рамках одного запроса, но вместо этого поддерживают гибридные *алгоритмы слияния поиска* (search fusion algorithms), которые позволяют вам объединять результаты из отдельных лексических и векторных запросов продуманным образом. В других случаях вы можете использовать векторную базу данных, которая отделена от вашей лексической поисковой системы, и в этом случае вы можете в конечном итоге использовать аналогичный алгоритм слияния для объединения этих результатов вне системы.

15.5.1. Алгоритм RRF

В этом разделе мы продемонстрируем популярный гибридный метод поиска для объединения двух наборов результатов поиска, называемый *алгоритмом RRF* (взаимным ранговым слиянием, англ. *reciprocal rank fusion*). RRF объединяет два или более наборов результатов поиска, ранжируя документы на основе их относительных рангов в различных наборах результатов. Алгоритм прост: для каждого документа он суммирует обратные значения рангов документа в каждом наборе результатов, а затем сортирует документы на основе этой суммы. Этот алгоритм особенно эффективен, когда два набора результатов поиска являются взаимодополняющими, так как он ранжирует выше те документы, которые уже ранжированы выше в любом из наборов, но выше всего, когда они ранжированы в обоих наборах. Следующий листинг реализует алгоритм RRF.

Листинг 15.17. RRF для объединения нескольких наборов результатов поиска

```

search_results – это список наборов ранжированных
документов, связанных с различными поисками
(лексический поиск, векторный поиск и т. д.).

def reciprocal_rank_fusion(search_results, k=None):
    if k is None: k = 60
    scores = {}
    for ranked_docs in search_results:
        for rank, doc in enumerate(ranked_docs, 1):
            scores[doc["id"]] = (scores.get(doc["id"], 0) +
                                (1.0 / (k + rank)))
    sorted_scores = dict(sorted(scores.items(),
                                key=lambda item: item[1], reverse=True))
    return sorted_scores

ranked_docs – это список до-
кументов для
определенного
поиска.

Рейтинг до-
кумента увели-
чивается, если
он находится
в несколь-
ких списках
ranked_docs
и выше в каж-
дом списке.

Возвращает документы,
отсортированные от са-
мого высокого рейтинга
RRF к самому низкому.

```

В этой функции передается произвольный список (`search_results`) наборов ранжированных документов из каждого отдельного поиска. Например, у вас может быть два предмета в `search_results`:

- ранжированные документы из лексического поиска,
- ранжированные документы из векторного поиска.

Затем оценка RRF для каждого документа вычисляется путем суммирования обратных рангов документа ($1 / (k + \text{rank})$) из каждого набора результатов поиска (`ranked_docs`). Далее окончательные оценки RRF сортируются и возвращаются.

Параметр `k` – это константа, которую можно увеличить, чтобы не допустить, чтобы выброс, ранжированный высоко в одном результате поиска, имел слишком большой вес. Чем выше `k`, тем больший вес дается документам, которые появляются в нескольких ранжированных списках документов, в отличие от документов, которые ранжируются выше в любом заданном списке ранжированных документов. Параметр `k` часто устанавливается на 60 по умолчанию, основываясь на исследованиях, показывающих, что это значение хорошо работает на практике (<https://plg.uwaterloo.ca/~gvcormac/cormacksigir09-rrf.pdf>).

Мы интегрировали логику RRF в функцию `collection.hybrid_search`, которую вызовем в листинге 15.19. Однако сначала давайте посмотрим, как выглядят начальные результаты для лексического поиска по сравнению с соответствующим векторным поиском для примера запроса фразы: "singin' in the rain".

Листинг 15.18. Выполнение лексического и векторного поиска независимо

```

over_request_limit = 15
base_query = {"return_fields": ["id", "title", "id", "image_id",
                                "movie_id", "score", "image_embedding"],

```



```

        "limit": over_request_limit,
        "order_by": [("score", "desc"), ("title", "asc")]}

def lexical_search_request(query_text):
    return {"query": query_text,
            "query_fields": ["title", "overview"],
            "default_operator": "OR",
            **base_query}

def vector_search_request(query_embedding):
    return {"query": query_embedding,
            "query_fields": ["image_embedding"],
            "quantization_size": "FLOAT32",
            **base_query}

def display_lexical_search_results(query_text):
    collection = engine.get_collection("tmdb_lexical_plus_embeddings")
    lexical_request = lexical_search_request(query_text)
    lexical_search_results = collection.search(**lexical_request)

    display_results(lexical_search_results, display_header= \
        get_display_header(lexical_request=lexical_request))

def display_vector_search_results(query_text):
    collection = engine.get_collection("tmdb_lexical_plus_embeddings")
    query_embedding = encode_text(query_text)
    vector_request = vector_search_request(query_embedding)
    vector_search_results = collection.search(**vector_request)

    display_results(vector_search_results, display_header= \
        get_display_header(vector_request=vector_request))

query = ''' + "singin' in the rain" + '''
display_lexical_search_results(query)
display_vector_search_results(query)

```

Этот листинг берет запрос лексической фразы "singin' in the rain" и кодирует его как эмбединг с помощью той же функции `encode_text` из листинга 15.15. Затем он выполняет лексический поиск и векторный поиск по коллекции `tmdb_lexical_plus_embeddings`, которая содержит как текстовые поля, необходимые для лексического поиска, так и эмбединг изображений для некоторых фильмов в наборе данных TMDb. Результаты лексического поиска и векторного поиска показаны на рис. 15.10 и 15.11 соответственно.

Лексический запрос: "singin' in the rain"

Singin' in the Rain
(Score: 6.5101776)



Рис. 15.10. Один лексический результат поиска для фразового запроса «singin' in the rain»

В этом случае лексический поиск возвращает только один результат: фильм «Поющие под дождем». Поскольку запрос пользователя был фразовым (заключенным в кавычки) запросом для точного названия заголовка, он идеально совпал и нашел только конкретный предмет, который, вероятно, искал пользователь. На рис. 15.11 показаны соответствующие результаты поиска векторов для того же запроса.

Векторный запрос: [0.048921154115761124, 0.024482925930009714, ...]

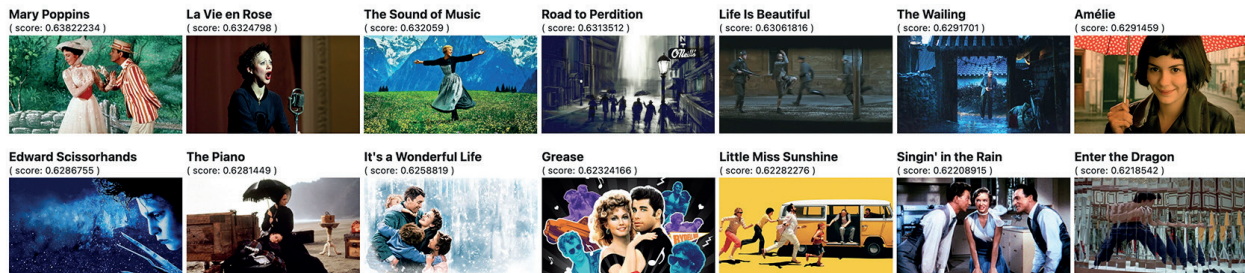


Рис. 15.11. Результаты векторного поиска для запроса "singin' in the rain "

Вы заметите, что результаты поиска векторов обычно показывают изображения, содержащие дождь или пение или соответствующие похожим понятиям, таким как зонтики, погода и мюзиклы. Это связано с тем, что наш поиск векторов выполняется по эмбедингам изображений, поэтому результаты основаны на визуальном содержании изображений, а не на текстовом содержании фильмов. Хотя эти изображения понятийно соответствуют значению слов в запросе лучше, чем результаты лексического поиска, они не содержат точного соответствия названий, как результаты лексического поиска, поэтому идеальный документ для фильма «Поющие под дождем» отсутствует в нескольких верхних результатах.

Объединив эти два набора результатов с помощью RRF, мы можем получить лучшее из обоих наборов: точное соответствие названий из лексического поиска и понятийно релевантные изображения из векторного поиска. В следующем листинге мы показываем, как объединить эти два набора результатов поиска, вызвав `collection.hybrid_search`.

Листинг 15.19. Функция гибридного поиска

```
def display_hybrid_search_results(text_query, limit=8):
    lexical_search = lexical_search_from_text_query(text_query)
    vector_search = vector_search_from_embedding(encode_text(text_query))
    hybrid_search_results = collection.hybrid_search(
        [lexical_request, vector_request], limit=10,
        algorithm="rrf", algorithm_params={"k": 60})
    display_header = get_display_header(lexical_search, vector_search)
    display_results(hybrid_search_results, display_header)

display_hybrid_search_results(query)
```

Мы передаем массив поисковых запросов для выполнения – в данном случае наши `lexical_search` и `vector_search` из листинга 15.18. Функция `collection.hybrid_search` внутренне по умолчанию использует алгоритм RRF

с установленным $k=60$, поэтому, если вы хотите явно передать эти параметры или изменить их, полный синтаксис выглядит следующим образом:

```
collection.hybrid_search([lexical_search, vector_search], limit=10,
                        algorithm="rrf", algorithm_params={"k": 60})
```

Вызов выполняет оба поиска и объединяет результаты с использованием RRF, вывод показан на рис. 15.12.

Результаты гибридного поиска:

--Лексический запрос: «singing in the rain»

--Векторный запрос: [0.048921154115761124, 0.024482925930009714, ...]

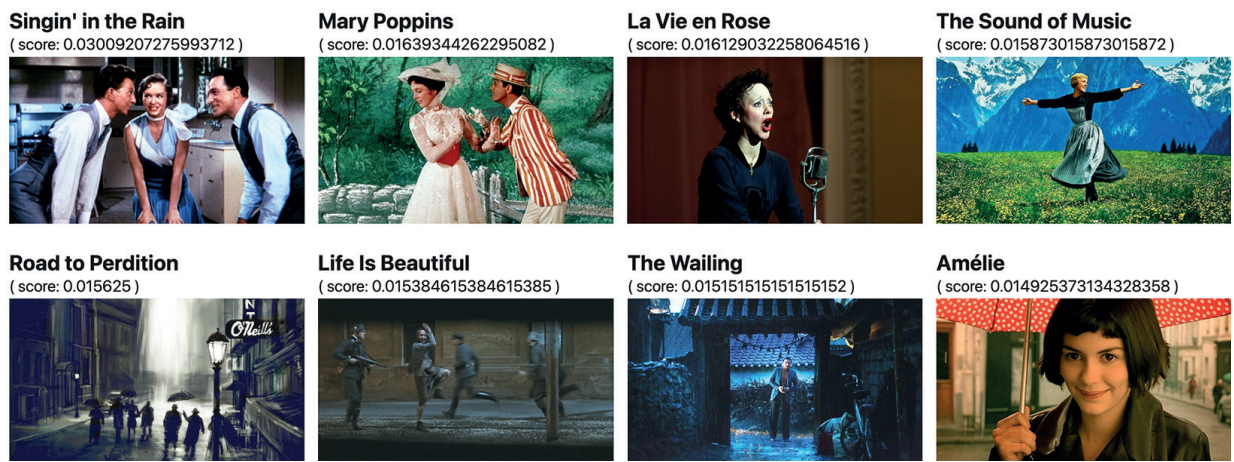


Рис. 15.12. Результаты гибридного поиска для лексического и векторного поиска с использованием RRF для запроса "singing in the rain"

Эти результаты демонстрируют, как гибридный поиск часто может предоставить лучшее из обоих методов: точные совпадения ключевых слов из лексического поиска и понятийно релевантные результаты из векторного поиска. Векторный поиск часто испытывает трудности с сопоставлением точных имен, конкретных ключевых слов и названий продуктов или идентификаторов. Аналогично лексический поиск пропускает понятийно релевантные результаты для текста запроса. В этом случае верхний результат – это точное совпадение, которое пользователь, вероятно, искал (ранее отсутствовало в результатах векторного поиска), но теперь оно дополнено другими результатами, понятийно похожими на запрос пользователя (ранее отсутствовало в лексическом поиске).

Конечно, вы можете передать более двух поисков в функцию гибридного поиска, если хотите. Например, вы можете добавить дополнительное векторное поле, содержащее эмбединг для заголовка или обзора, чтобы вместо просто лексического поиска по текстовому контенту и векторного поиска по изображениям у вас также был семантический поиск по текстовому контенту. Использование RRF с этими тремя наборами результатов поиска позволит вам объединить лучшее из всех трех подходов к поиску в один набор результатов.

Помимо преодоления соответствующих функциональных ограничений поиска по ключевым словам и векторного поиска, гибридный по-

иск часто дает лучший общий рейтинг поиска. Это связано с тем, что алгоритмы, такие как RRF, разработаны для ранжирования документов на самом высоком уровне, когда они находятся в нескольких наборах результатов поиска. Когда одни и те же предметы возвращаются как релевантные из разных поисков, алгоритм слияния может использовать эту согласованность между наборами результатов для повышения оценок этих предметов. Это особенно полезно, когда один или несколько наборов результатов поиска содержат нерелевантные документы, поскольку согласие между двумя наборами результатов может помочь отфильтровать шум и вывести наиболее релевантные результаты. Листинг 15.20 демонстрирует эту концепцию с использованием запроса *the hobbit*.

Листинг 15.20. Лексический, векторный и гибридный поиск *the hobbit*

```
query = "the hobbit"
display_lexical_search_results(query)
display_vector_search_results(query)
display_hybrid_search_results(query)
```

На рис. 15.13–15.15 показаны результаты из листинга 15.20.

На рис. 15.13 вы увидите пять релевантных результатов в первых шести результатах («Хоббит» является частью франшизы «Властелин колец»). Текст «*the Hobbit*» появляется в названии трех из первых четырех результатов. На пятой позиции результатов лексического поиска находится один явно плохой результат, а все результаты – плохие после шестого документа, в основном просто совпадающие по ключевым словам, таким как «*the*» и «*of*» в полях *title* и *overview*.

Лексический запрос: *the Hobbit*

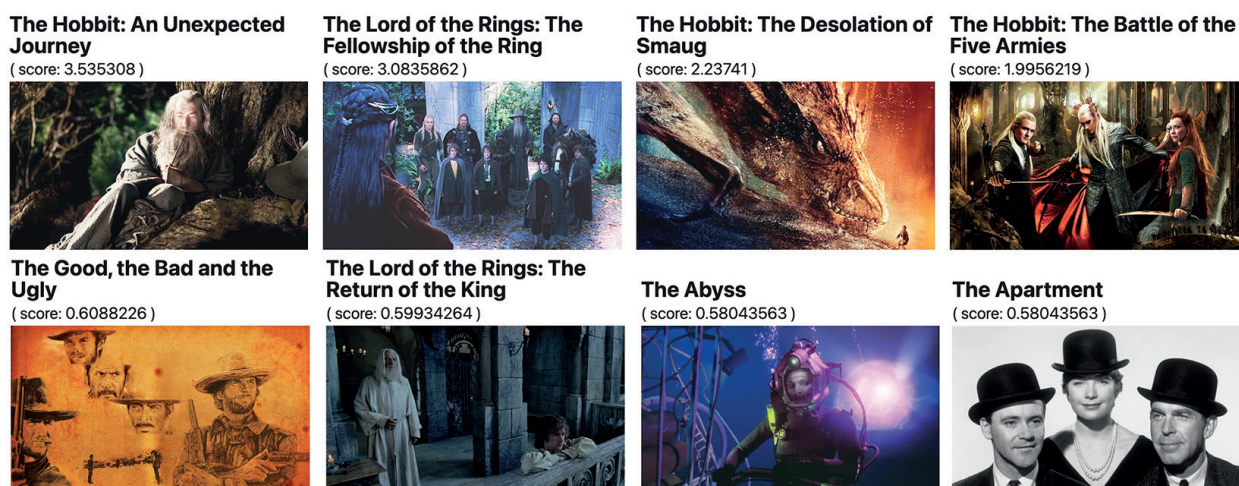


Рис. 15.13. Результаты лексического поиска для запроса *the Hobbit*

На рис. 15.14 показаны результаты векторного поиска для того же запроса. Они также показывают пять релевантных результатов, связанных с «Властелином колец», но один из релевантных результатов

лексического поиска отсутствует, и дополнительный релевантный фильм был возвращен в результатах векторного поиска.

Векторный запрос: $[-0.016278795212303375, -0.0143564000471339553, \dots]$

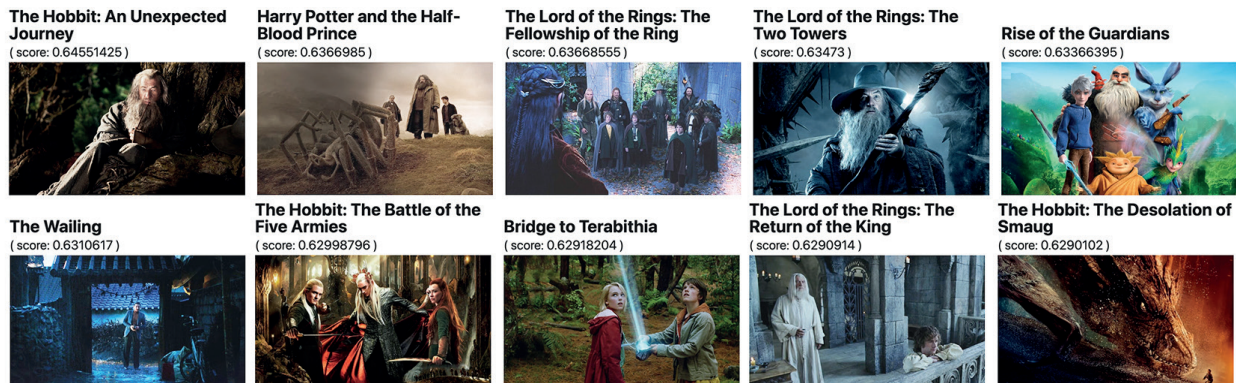


Рис. 15.14. Результаты векторного поиска для запроса the Hobbit

Многие из оставшихся результатов понятийно связаны с запросом, показывая в основном фильмы с фантастическими пейзажами и магией. Учитывая сильное перекрытие хороших результатов между результатами лексического и векторного поиска, а также отсутствие перекрытия между менее релевантными результатами, мы должны ожидать, что результаты гибридного поиска будут довольно хорошими.

Действительно, на рис. 15.15 мы видим, что все первые пять результатов релевантны и что шесть из первых семи результатов связаны с «Властелином колец». Мы перешли от двух разных списков результатов поиска, в каждом из которых отсутствовали разные документы и показывались некоторые нерелевантные результаты, к окончательному списку результатов.

Результаты гибридного поиска:

--Лексический запрос: the Hobbit

--Векторный запрос: $[-0.016278795212303375, -0.0143564000471339553, \dots]$

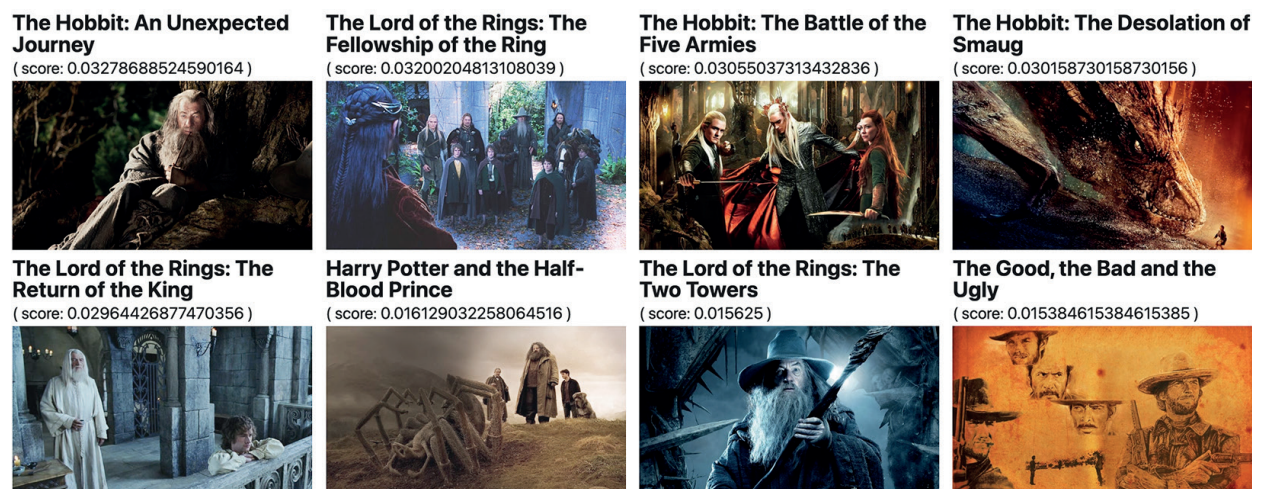


Рис. 15.15. Результаты гибридного поиска для результатов лексического и векторного поиска с использованием RRF для запроса the Hobbit

RRF позволил нам вывести на поверхность отсутствующие результаты из обоих списков и опустить нерелевантные результаты, появляющиеся только в одном списке, действительно подчеркивая лучшие качества каждой парадигмы поиска (лексической против векторной), преодолевая при этом каждую из их слабостей.

15.5.2. Другие алгоритмы гибридного поиска

Хотя RRF является популярным и эффективным алгоритмом для объединения результатов поиска, конечно, есть много других алгоритмов, которые можно использовать для аналогичных целей. Одним из популярных алгоритмов является *алгоритм RSF* (слияние относительных оценок, англ. *relative score fusion*), который похож на RRF, но использует относительные оценки документов в каждом наборе результатов поиска для объединения результатов. Поскольку оценки релевантности часто не сопоставимы между различными алгоритмами ранжирования, оценки для модальности запроса обычно масштабируются до одного и того же диапазона (часто от 0.0 до 1.0) на основе минимальных и максимальных оценок от каждой модальности. Затем относительные оценки объединяются с использованием взвешенного среднего, при этом веса часто устанавливаются на основе относительной характеристики каждой модальности в наборе проверки.

Другой распространенный способ объединения результатов поиска – использовать одну модальность для начального соответствия, а затем переранжировать результаты с использованием другой модальности. Многие системы поддерживают как лексический поиск, так и векторный поиск как отдельные операции, но позволяют вам запускать одну версию запроса (например, лексический запрос), а затем переранжировать полученные документы с использованием другой версии (например, векторного запроса).

Например, если вы хотите убедиться, что у вас есть соответствие по определенным ключевым словам, но также хотите повысить результаты, которые понятийно похожи на запрос, вы можете сначала запустить лексический поиск, а затем переранжировать результаты с использованием векторного поиска. Следующий листинг демонстрирует эту процедуру.

Листинг 15.21. Гибридный лексический поиск с перанкингом векторного поиска

```
def lexical_vector_rerank(text_query, limit=10):
    lexical_request = lexical_search_request(text_query)
    vector_request = vector_search_request(encode_text(text_query))
    hybrid_search_results = collection.hybrid_search(
        [lexical_request, vector_request],
        algorithm="lexical_vector_rerank", limit=limit)
    header = get_display_header(lexical_request, vector_request)
    display_results(hybrid_search_results, display_header=header)

lexical_vector_rerank("the hobbit")
```


Внутренне это преобразуется в обычный лексический поиск, но с результатами, пересортированными с использованием оценки векторного поиска для того же (закодированного) запроса. Синтаксически это ключевые части поискового запроса, сгенерированного листингом 15.21:

```
{'query': 'the hobbit',
 'query_fields': ['title', 'overview'],
 'default_operator': 'OR',
 ...
 'order_by': [('score', 'desc'), ('title', 'asc')],
 'rerank_query': {
   'query': [-0.016278795212303375, ..., -0.02110762217111629],
   'query_fields': ['image_embedding'],
   'quantization_size': 'FLOAT32', ...
   'order_by': [('score', 'desc'), ('title', 'asc')], ...
 }
}
```

Вывод этого лексического поиска с реранкингом векторного поиска показан на рис. 15.16. В этом конкретном случае результаты выглядят очень похожими на результаты RRF. Обратите внимание, что, в отличие от рис. 15.10, где оператор по умолчанию для лексического запроса был AND, что привело к одному точному лексическому совпадению, здесь оператор по умолчанию является OR, так что для векторного поиска для реранкинга возвращается больше результатов. Если вы хотите повысить точность и возвращать только более точные лексические совпадения, можете установить оператор по умолчанию на AND или добавить пороговое значение `min_match` в запрос лексического поиска.

Результаты гибридного поиска:

--Лексический запрос: the Hobbit

--Векторный запрос: [-0.016278795212303375, -0.0143564000471339553, ...]

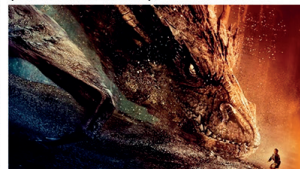
The Hobbit: An Unexpected Journey
(score: 4.1808224)



The Lord of the Rings: The Fellowship of the Ring
(score: 3.7202718)



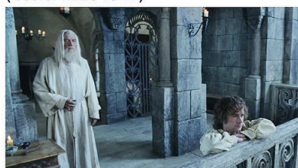
The Hobbit: The Desolation of Smaug
(score: 2.8664203)



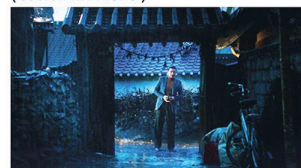
The Hobbit: The Battle of the Five Armies
(score: 2.6256099)



The Lord of the Rings: The Return of the King
(score: 1.2284341)



The Wailing
(score: 1.2114973)



The Lord of the Rings: The Two Towers
(score: 1.2099149)



Rise of the Guardians
(score: 1.0686013)



Рис. 15.16. Гибридный поиск с помощью векторного реранкинга лексического запроса

Функционально лексический поиск полностью отвечает за то, какие документы возвращаются, в то время как векторный поиск полностью отвечает за порядок результатов. Это может быть предпочтительным, если вы хотите подчеркнуть лексическое соответствие ключевых слов, но с более семантически релевантным ранжированием. С другой стороны, подход слияния, такой как RRF или RSF, обеспечит более смешанный подход, гарантируя, что будут обнаружены лучшие результаты из каждой модальности. Существует множество других способов объединения результатов поиска, и наилучший подход будет зависеть от конкретного варианта использования и относительных сильных и слабых сторон каждой модальности поиска.

15.6. Конвергенция контекстных технологий

Точно так же, как рекомендации, чат-боты и системы вопрос-ответ являются типами поиска информации с использованием ИИ, многие другие технологии также начинают конвергенцию с поисковыми системами. Появляются векторные базы данных, которые функционируют как поисковые движки для мультимодальных данных, и многие различные источники данных и модальности данных объединяются в качестве дополнительных контекстов для оптимального сопоставления, ранжирования и рассуждения по данным. Генеративные модели обеспечивают базовое понимание, достаточное для создания новых письменных и художественных произведений.

Но большинство этих технологий по-прежнему внедряются в производственные системы по частям. Поскольку исследователи продолжают работать над созданием общего искусственного интеллекта, интеллектуальных роботов и более интеллектуальных поисковых систем, мы, вероятно, увидим продолжающуюся конвергенцию технологий, интегрирующих новые и разные контексты для создания более полного понимания контента, доменов и пользователей. Рисунок 15.17 демонстрирует, как эта конвергенция технологий, скорее всего, будет выглядеть в ближайшие годы.

На этом рисунке мы видим текстовые, аудио- и видеомодальности, которые мы уже обсуждали, а также модальность метаданных, которая может предоставлять дополнительный контекст. Мы видим сенсорную модальность, которая может быть доступна для физической сети устройств, оснащенных датчиками, или робота с прямым интерактивным доступом к физическому миру. Мы также видим временную модальность, поскольку время как запросов, так и искомых данных может влиять на релевантность результатов.

Точно так же, как модели LLM изучают как домен, так и языковую структуру текста (см. главу 13), дополнительные культурные контексты и обычаи могут быть изучены с помощью базовых моделей, основанных на географии и наблюдаемом поведении из взаимодействий в реальном мире.

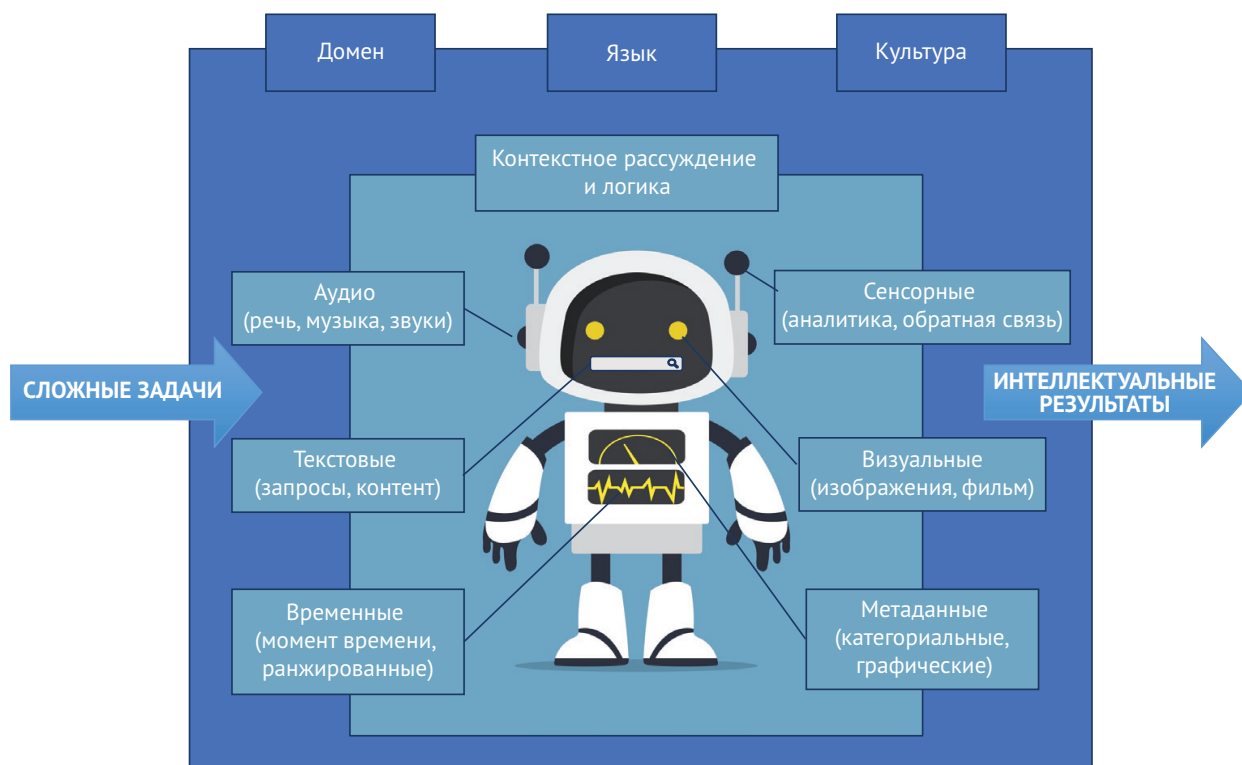


Рис. 15.17. Конвергенция контекстных технологий для предоставления более интеллектуальных результатов

Многое из этого может показаться *очень далеким* от традиционного поиска или даже поиска на основе ИИ сегодня – и это так! Однако цель поиска – наилучшим образом удовлетворить информационную потребность пользователя, и по мере появления все большего количества пересекающихся технологий для решения этой проблемы с разных сторон мы видим, что поиск становится критически важным базовым слоем, с которым будут интегрироваться многие из этих технологий.

Многие специалисты по ИИ уже осознают важность использования RAG для поиска и предоставления контекста в реальном времени для генеративных моделей ИИ, а соображения безопасности данных, точности данных и размера модели почти наверняка делают поисковые системы критически важной основой для систем генеративного ИИ на основе знаний в будущем. Большие объемы данных, поступающие из многих источников, должны быть проиндексированы и доступны для извлечения и использования в реальном времени генеративными моделями ИИ.

Объединение этих технологий в сквозные системы позволит решать эти проблемы поиска на основе ИИ лучше, чем любая из этих систем могла бы сделать по отдельности. Будет ли это называться «робототехникой», «ИИ», «общим интеллектом», «поиском на основе ИИ» или чем-то еще, нам предстоит увидеть. Но многие из методов поиска на основе ИИ, которые вы изучили, были и будут оставаться центральными для разработки этих систем контекстного рассуждения следующего поколения.

15.7. Все вышеперечисленное, пожалуйста!

В этой книге мы глубоко погрузились в основные понятия построения поиска на основе ИИ. Мы рассмотрели как теорию, лежащую в основе того, как работает современная релевантность поиска на основе ИИ, так и примеры кода, демонстрирующие каждую тему с реальными вариантами использования.

Методы и алгоритмы, которые мы рассмотрели, не будут применимы ко всем вариантам использования, но обычно, объединяя несколько подходов, вы сможете обеспечить более эффективный и релевантный поиск.

Хотя область поиска на основе ИИ быстро развивается, особенно из-за частых инноваций, происходящих с моделями генеративного фундамента и плотными векторными подходами к поиску, основные принципы остаются прежними. Задача поисковой системы заключается в том, чтобы понимать намерения пользователя и возвращать контент, который наилучшим образом удовлетворяет информационные потребности каждого пользователя. Это требует правильного понимания контекста контента, контекста пользователя и контекста домена для каждого взаимодействия. Помимо изучения языковых моделей и графов знаний из вашего контента, возможности вашей поисковой системы значительно возрастут, если она сможет неявно учиться у ваших пользователей, посредством их взаимодействия (пользовательских сигналов) с поисковой системой.

Вы узнали, как работают ключевые алгоритмы поиска на основе ИИ и как внедрить и автоматизировать эти алгоритмы в самообучающуюся поисковую систему. Независимо от того, нужно ли вам автоматизировать создание моделей бустинга сигналов, научиться ранжировать модели, создать модель кликов, модель совместной фильтрации, граф знаний или тонко настроенную модель трансформера на основе глубокого обучения для улучшенного семантического поиска и ответов на вопросы, теперь у вас есть знания и навыки, необходимые для реализации поисковой системы на основе ИИ мирового класса. Мы с нетерпением ждем, что вы создадите!

Резюме

- Базовые модели являются начальными моделями, которые обучаются и впоследствии могут быть тонко настроены для определенных доменов или задач.
- Инжиниринг подсказок позволяет вам вводить дополнительный контекст и данные в каждый запрос, предоставляя способ выполнять тонкую настройку контента, который будет возвращен для запроса, в режиме реального времени. Приложения на основе LLM в идеале должны быть запрограммированы на автоматическую генерацию и оптимизацию подсказок, а не на ручную проверку и корректировку подсказок, чтобы они могли автоматически обрабатывать изменения модели и среды с течением времени, при этом оставаясь оптимальными.

- Суммаризация результатов поиска и генерация обучающих данных – две ключевые области, в которых базовые модели могут помочь повысить релевантность поисковых систем.
- Совместное обучение эмбединговых моделей на нескольких типах данных обеспечивает мощные возможности мультимодального поиска (текст в изображение, изображение в изображение, гибридный поиск текст плюс изображение в изображение и т. д.), расширяя возможности пользователя и способность поисковой системы интерпретировать намерения пользователя.
- Поиск на основе ИИ быстро развивается с появлением больших языковых моделей, плотного векторного поиска, мультимодального поиска, разговорного и контекстного поиска, генеративного поиска и новых методов гибридного поиска.
- Существует множество методов реализации поиска на основе искусственного интеллекта, и лучшими системами будущего будут те, которые смогут эффективно применять несколько релевантных подходов в гибридных поисковых системах.

Приложение А.

Запуск примеров кода

Во время вашего путешествия по стране поиска на основе ИИ мы пройдемся по большому количеству кода и запустим примеры программного обеспечения, демонстрирующие методы, о которых рассказано в этой книге. В этом приложении показано, как легко настроить и запустить сопутствующий исходный код, чтобы вы могли экспериментировать с живыми, работающими примерами по мере изучения материала. Мы рассмотрим, как упаковывается исходный код книги, как извлекать и собирать исходный код, а также как работать с блокнотами Jupyter и Docker для запуска примеров.

А.1. Общая структура примеров кода

Создание поисковой системы на основе ИИ требует интеграции множества компонентов и библиотек. В качестве нашей поисковой системы по умолчанию мы будем использовать Apache Solr, который внутренне опирается на Apache ZooKeeper. Вы также можете заменить Solr многими другими популярными поисковыми системами и векторными базами данных – см. инструкции в приложении В.

Для важных задач обработки данных и машинного обучения мы используем Apache Spark. Мы используем Python в качестве языка программирования для всех примеров кода и полагаемся на множество зависимостей – внешних модулей и библиотек Python – в дополнение к другим системным зависимостям (например, Java), которые требуются некоторым нашим системам. Конечно, потребуется запускать наши примеры кода и видеть результаты в удобном для пользователя виде, что мы делаем с помощью блокнотов Jupyter.

Вместо того чтобы устанавливать десятки библиотек программного обеспечения и сотни зависимостей, чтобы все это работало, мы мак-

симально упростили этот процесс, упаковав все примеры книги в контейнеры Docker, которые уже полностью настроены и готовы к использованию. Это означает, что перед запуском примеров кода в этой книге необходимо установить только одно приложение: Docker.

Docker позволяет создавать и запускать крошечные контейнеры – полностью функционирующие виртуальные машины, которые работают только под управлением облегченной операционной системы со всем необходимым программным обеспечением и зависимостями, уже установленными и настроенными. Это позволяет запускать код в большинстве операционных систем (macOS, Windows, Linux) без необходимости настраивать какие-либо системные зависимости. После запуска всех служб все листинги кода в книге будут доступны через блокноты Jupyter, которые будут служить интерфейсом для запуска и экспериментирования с примерами кода, чтобы увидеть полученные результаты.

A.2. Извлечение исходного кода

Исходный код, прилагаемый к этой книге, доступен на GitHub здесь: <https://github.com/treygrainger/ai-powered-search>. Чтобы извлечь код, используйте установленный клиент Git или откройте интерфейс командной строки в предпочитаемой вами папке разработки и выполните следующую команду:

```
git clone https://github.com/treygrainger/ai-powered-search.git
```

Теперь у вас должна быть новая папка в текущем каталоге с именем `ai-powered-search/`, которая содержит весь исходный код для книги.

Если у вас не установлен Git или вы не можете извлечь код с помощью предыдущей команды, на сайте также есть возможность загрузить исходный код в виде zip-файла через веб-браузер, который затем вам нужно будет просто распаковать в папку разработки.

При желании вы можете переименовать или переместить папку `ai-powered-search/`; в остальной части этой книги мы будем просто ссылаться на этот каталог с помощью переменной `$AIPS_HOME`.

A.3. Сборка и запуск кода

Как уже упоминалось ранее, Docker – это единственная ключевая зависимость, которую вы должны установить в своей системе для сборки и запуска примеров кода нашей книги «Поиск на основе искусственного интеллекта». Мы не будем рассматривать этот процесс установки здесь, поскольку он зависит от системы и время от времени меняется. Пожалуйста, посетите <https://docker.com> для получения инструкций по загрузке и установке. Следует также отметить, что вместо Docker можно использовать другие инструменты управления контейнерами, такие как Podman, хотя мы не будем рассматривать их здесь.

После установки Docker обязательно измените каталог в интерфейсе командной строки на каталог \$AIPS_HOME (`cd $AIPS_HOME`). Чтобы построить и запустить кодовую базу, вам просто нужно выполнить следующую команду:

```
docker compose up
```

Обратите внимание, что эта команда эквивалентна запуску `docker compose up solr`. Кодовая база книги работает с несколькими поисковыми системами и векторными базами данных, и этот (опциональный) последний аргумент позволяет вам указать, какой движок (или какие движки) вы хотите использовать (по умолчанию используется Apache Solr). Например, если хотите использовать OpenSearch с книгой, вы должны запустить `docker compose up opensearch`. Для получения дополнительной информации о других поддерживаемых поисковых системах см. приложение В.

СОВЕТ. Команда `docker compose up` запускается на переднем плане вашей консоли, что позволяет вам видеть все журналы, транслируемые в режиме реального времени, но это также означает, что все ваши контейнеры будут уничтожены, если вы закроете консоль. Если вы хотите вместо этого запустить контейнеры в фоновом режиме и закрыть или продолжить использовать консоль, можете передать параметр `-d` или `--detach` (`docker compose up -d`). Если вы запускаете ее таким образом, обязательно явно запустите `docker compose -profile all down`, когда закончите, чтобы остановить бесконечную работу контейнеров в фоновом режиме, потребляя ресурсы.

Эта команда займет некоторое время для выполнения в первый раз, так как она извлекает весь код, операционные системы и зависимости, необходимые для сборки и запуска программного обеспечения, сопровождающего эту книгу. Однако после завершения команды у вас будут все необходимые службы (Jupyter, Spark, ZooKeeper, Solr и т. д.), работающие в отдельных контейнерах Docker.

Теперь, чтобы начать, просто откройте веб-браузер и перейдите по адресу <http://localhost:8888>. Это перенаправит вас на экран приветствия со списком всех глав книги и соответствующих им блокнотов Jupyter (рис. А.1). Теперь вы можете кликнуть на любую из глав, чтобы запустить примеры кода.

По умолчанию блокноты Jupyter загружаются в среду Jupyter Labs, которая является средой разработки для работы с блокнотами, их отладки и навигации по ним. Она позволяет открывать и редактировать несколько блокнотов одновременно (в отдельных вкладках), а также перемещаться и изменять структуру каталогов в левой части экрана по мере исследования.

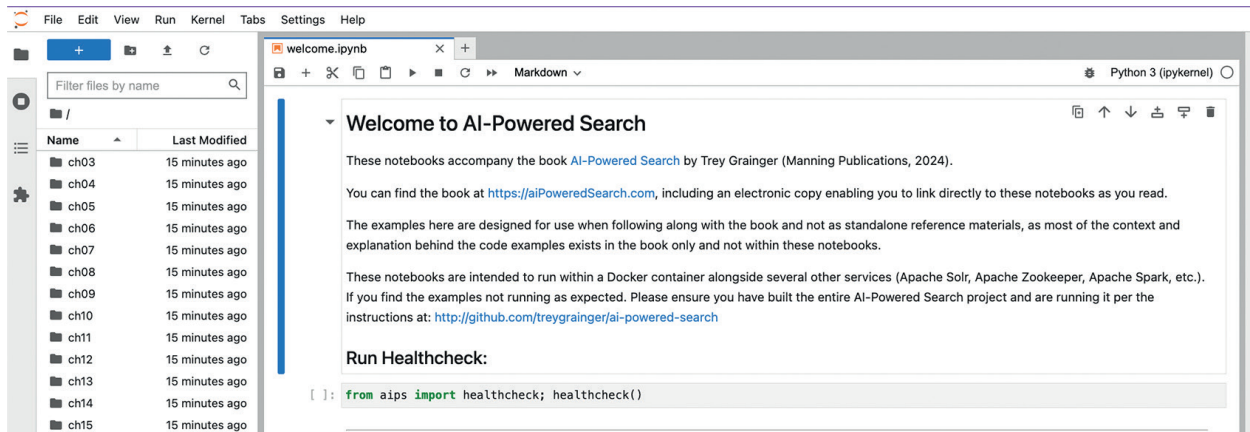


Рис. А.1. Экран приветствия. Как только вы это увидите, контейнеры «Поиск на основе искусственного интеллекта» будут уже созданы, а блокноты Jupyter уже запущены

Если вы просто хотите следовать книге в более простой среде, можете перейти по адресу <http://localhost:8888/notebooks> и перемещаться по оглавлению, чтобы следовать по одному блокноту за раз по мере продвижения по книге. Мы будем использовать этот более простой интерфейс в следующем разделе, хотя та же функциональность (и даже больше) присутствует в среде Jupyter Labs по умолчанию по адресу <http://localhost:8888>.

А.4. Работа с Jupyter

После просмотра блокнота `welcome.ipynb` вы увидите на экране несколько ячеек данных, включая вводное сообщение, скрипт «проверки работоспособности» и оглавление различных блокнотов, содержащих исполняемые примеры из книги.

Если вы никогда раньше не использовали Jupyter, это инструмент, который позволяет вам смешивать разметку (обычно инструкции и пояснения) и код в вашем браузере, а также редактировать, запускать и взаимодействовать с выводом из примеров кода. Это значительно упрощает обучение, так как вам не нужно использовать инструменты командной строки, и вместо этого вы можете полностью взаимодействовать с готовыми к выполнению примерами одним кликом кнопки.

Вы заметите панель инструментов в верхней части экрана (под строкой меню), которая позволяет вам взаимодействовать с разделами содержимого, называемыми *ячейками* (cells), в блокноте. Вы можете использовать эти инструменты для перемещения вверх и вниз, для остановки и перезапуска блокнота или для последовательного выполнения каждой ячейки с помощью кнопки **Run**.

На рис. А.2 клик по **Run**, когда выделена ячейка кода проверки работоспособности, приведет к выполнению проверки работоспособности для подтверждения того, что все контейнеры Docker запущены, а службы, работающие в них, работоспособны и отвечают.

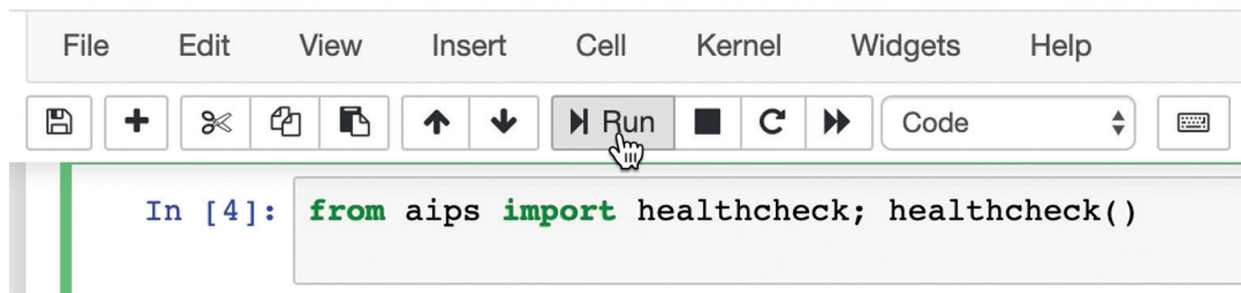


Рис. А.2. Запуск примеров кода. Клик Run на панели инструментов запустит все примеры в текущей ячейке (если таковые имеются) и перейдет к следующей ячейке

Рисунок А.3 показывает ответ, который вы увидите, когда все работает как ожидалось.

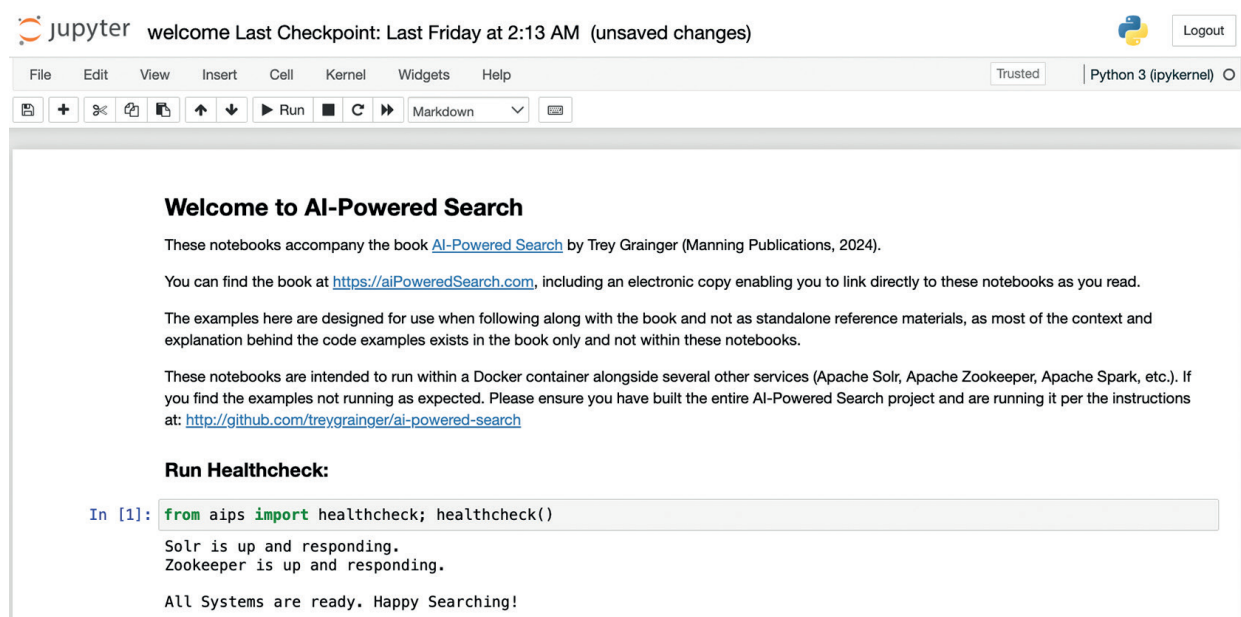


Рис. А.3. Успешное выполнение проверки работоспособности. Вы должны увидеть это сообщение, если все работает правильно

На этом этапе вы можете прокрутить экран вниз до оглавления и перейти к блокнотам для каждой главы. Конечно, поскольку объяснение примеров содержится в книге, вы, вероятно, предпочтете работать с примерами по мере чтения книги, чтобы иметь соответствующий фон при их запуске. Блокноты Jupyter не предназначены для использования в качестве отдельных примеров, поэтому вы, вероятно, захотите держать книгу под рукой, чтобы обеспечить контекст.

Все блокноты Jupyter разработаны так, чтобы быть независимо идемпотентными¹. Это означает, что, хотя все шаги в блокноте долж-

¹ Идемпотентность в программировании – это свойство некоторых операций, которое означает, что повторное выполнение операции не приводит к изменению состояния системы или ресурса. Идемпотентность помогает избежать непредсказуемых результатов при повторных операциях. Это упрощает написание программ, делая их более стабильными и предсказуемыми. – Прим. ред.

ны быть выполнены, чтобы гарантировать успешный результат, вы всегда можете начать заново с начала любого блокнота, и он «сбросится» до ожидаемых результатов, необходимых для успешного выполнения последующих шагов. Если вы когда-либо столкнетесь с ошибками в блокноте, просто вернитесь к первой ячейке на странице и снова просмотрите весь блокнот.

A.5. Работа с Docker

Хотя все в предыдущих разделах должно работать так, как и ожидалось, конечно, возможно, что вы можете столкнуться с проблемами по ходу дела. Наиболее вероятная проблема, с которой вы столкнетесь, – это сбой одного из ваших контейнеров Docker или службы, работающей внутри него. Также возможно, если вы вносите изменения в базовые данные или конфигурацию в одной из служб, например, что вы можете привести ее в неисправное состояние.

Когда это произойдет, вы всегда можете удалить свои контейнеры и начать заново. Для этого просто выполните эту команду из каталога \$AIPS_HOME:

```
docker compose --profile all down && docker compose up
```

Помните, что, если вы внесли какие-либо изменения в свои блокноты или код, вся проделанная вами работа будет потеряна при запуске `docker compose --profile all down`. В целом примеры разработаны как временные. Если вы хотите сохранить свою работу при пересборке контейнеров, можете изменить файл `docker-compose.yaml`, чтобы смонтировать папки на локальном диске или использовать тома данных Docker, маркированные как внешние, которые будут постоянными. Пожалуйста, обратитесь к документации Docker, если вы планируете вносить изменения, так как механизмы и API могут время от времени меняться.

Если вы когда-либо задумаете изменить примеры кода или конфигурацию, вам также может потребоваться пересборка образов Docker. Когда вы запускаете `docker compose up` в первый раз, он извлечет или построит ваши образы и запустит их, но не пересоберет с изменениями, внесенными после первой сборки. Чтобы пересобрать все перед перезапуском контейнеров Docker, вы можете вместо этого выполнить команду:

```
docker compose up --build
```

Если вы просто хотите временно остановить работающие контейнеры Docker и возобновить их работу позже, можете выполнить `docker compose --profile all stop` для остановки, а затем `docker compose start later` для возобновления. Это полезно, если вы хотите сохранить любые внесенные изменения без сброса к чистой версии блокнотов при следующем запуске. Это должно дать вам все необходимое для запуска всех блокнотов и кода в нашей книге «Поиск на основе искусственного интеллекта».

Приложение В. Поддерживаемые поисковые системы и векторные базы данных

Мы используем поисковую систему Apache Solr с открытым исходным кодом в качестве поисковой системы по умолчанию на протяжении всей книги для обеспечения согласованности, но все алгоритмы в кодовой базе разработаны для работы с широким спектром поисковых систем и векторных баз данных. С этой целью, за исключением случаев, когда для демонстрации точки требуется движок-специфичный синтаксис, мы реализовали функцию поиска с использованием универсального интерфейса движка, который позволяет вам легко заменить предпочитаемую вами поисковую систему или векторную базу данных. В этом приложении мы рассмотрим список поддерживаемых движков, как заменить движок по умолчанию и как основные абстракции (`engine` и `collection`) используются на протяжении всей книги.

В.1. Поддерживаемые движки

Список поддерживаемых движков будет со временем расти, но на момент публикации изначально поддерживаются следующие поисковые системы:

- `solr` – Apache Solr;
- `opensearch` – OpenSearch;
- `bonsai` – Bonsai;
- `weaviate` – Weaviate.

Чтобы увидеть полный, самый актуальный список всех поддерживаемых движков, посетите <https://aipoweredsearch.com/supported-engines>.

В.2. Динамическая замена движка

Обычно вы будете работать только с одной поисковой системой или векторной базой данных одновременно при запуске примеров кода книги. Чтобы использовать любую конкретную поисковую систему, вам просто нужно указать имя движка (как указано выше) при запуске Docker.

Например, OpenSearch можно запустить так:

```
docker compose up opensearch
```

Это запустит все необходимые контейнеры Docker, необходимые для запуска opensearch (или указанного вами движка), а также установит этот движок как активный в блокнотах Jupyter книги для использования во всех примерах кода.

Обратите внимание, что некоторые движки, такие как управляемый поиск и службы на основе API, не требуют дополнительных локальных контейнеров Docker, поскольку их службы размещены в другом месте. Кроме того, некоторые движки могут требовать дополнительных параметров конфигурации, таких как ключи API, удаленные адреса/URL, порты и т. д. Эти параметры можно задать в файле .env в корне проекта.

Если вы хотите использовать другой движок в любой момент, можете перезапустить контейнеры Docker и указать новый движок, который вы хотите использовать:

```
docker compose up bonsai
```

Если вы хотите запустить более одного движка одновременно для экспериментов, можете предоставить список движков, которые вы хотите запустить, в конце команды docker compose up:

```
docker compose up solr opensearch weaviate
```

Первый движок, на который вы ссылаетесь в своей команде docker compose up, будет установлен как ваш активный движок в блокнотах Jupyter, а остальные будут доступны в режиме ожидания.

Если вы хотите переключиться на один из резервных движков в ваших живых блокнотах Jupyter (например, на opensearch), можете сделать это в любое время, просто выполнив следующую команду в любом блокноте:

```
import aips
aips.set_engine("opensearch")
```

Помните, что, если вы вызовете set_engine для движка, который в данный момент не запущен, это позже приведет к ошибкам, если

этот движок все еще недоступен, когда вы попытаетесь его использовать.

Вы также можете проверить текущий установленный движок в любое время, выполнив следующую команду:

```
aips.get_engine().name
```

В.3. Абстракции движка и коллекции

Индустрия поисковых систем полна различных терминов и понятий, и мы постарались абстрагироваться от этого как можно больше в кодовой базе. Большинство поисковых систем начинали с лексического поиска по ключевым словам и с тех пор добавили поддержку векторного поиска, а многие векторные базы данных начинали с векторного поиска и с тех пор добавили лексический поиск. Для наших целей мы просто думаем обо всех этих системах как о *движках сопоставления и ранжирования* и используем термин *движок* (поисковая система) для обозначения их всех.

Аналогично у каждого движка есть один или нескольких логических разделов или контейнеров для добавления данных. В Solr и Weaviate эти контейнеры называются *коллекциями*; в OpenSearch, Elasticsearch и Redis они называются *индексами*; а в Vespa они называются *приложениями*. В MongoDB исходные данные хранятся в *коллекциях*, но затем их можно скопировать в *индекс* для целей поиска. Наименования в разных движках также различаются.

Для правильной абстракции мы всегда используем термин *коллекция* в кодовой базе, поэтому каждый реализованный движок имеет интерфейс `collection`, через который можно запрашивать или добавлять документы.

Общедоступные методы в интерфейсе `engine` включают:

- `engine.create_collection(collection_name)` – создает новую коллекцию;
- `engine.get_collection(collection_name)` – возвращает существующую коллекцию.

Общедоступные методы в интерфейсе `collection` включают:

- `collection.search(**request)` – запускает поиск и возвращает результаты. Отдельные параметры запроса должны быть переданы как ключевые аргументы Python, например: `collection.search(query="keyword", limit=10)`;
- `collection.add_documents(docs)` – добавляет список документов в коллекцию;
- `collection.write(dataframe)` – записывает каждую строку из фрейма данных Spark в коллекцию как документ;
- `collection.commit()` – обеспечивает сохранение и доступность для поиска недавно добавленных документов.

Интерфейсы `engine` и `collection` также внутренне реализуют определения схем¹ и управление для всех наборов данных, используемых в книге.

Поскольку метод `collection.write` принимает тип данных датафрейм, мы используем помощников по мере необходимости при загрузке данных из дополнительных источников данных, таких как CSV или SQL:

- `collection.write(from_csv(csv_file))` – записывает каждую строку в файле CSV в коллекцию как документ;
- `collection.write(from_sql(sql_query))` – запускает запрос SQL и записывает каждую возвращенную строку в коллекцию как документ.

Для загрузки из этих дополнительных источников данных не требуется никакой дополнительной реализации, специфичной для движка, поскольку любой источник данных, который может быть сопоставлен с фреймом данных Spark, поддерживается неявно.

В.4. Добавление поддержки для дополнительных движков

Хотя мы надеемся в конечном итоге поддерживать большинство основных поисковых систем и векторных баз данных, вы можете обнаружить, что ваш любимый движок в настоящее время не поддерживается. Если это так, мы рекомендуем вам добавить его поддержку и отправить запрос на извлечение в кодовую базу. Интерфейсы `engine` и `collection` разработаны для простоты реализации, и вы можете использовать реализацию `solr` по умолчанию или любые другие уже реализованные движки в качестве справки.

Не все хранилища данных полностью поддерживают все возможности, реализованные в поиске на основе ИИ. Например, чистая векторная база данных может не поддерживать лексическое сопоставление ключевых слов и ранжирование, а некоторые поисковые системы могут не поддерживать векторный поиск. Аналогично некоторые специализированные возможности могут быть доступны только в определенных движках.

В то время как движок `solr` по умолчанию поддерживает все возможности поиска на основе ИИ, реализованные в книге, другим движкам могут потребоваться обходные пути, интеграция дополнительных библиотек или делегирование некоторых возможностей другому движку для определенных алгоритмов. Например, большинство движков не имеют встроенной поддержки семантических графов знаний и текстовых тегов, поэтому многие реали-

¹ XML Schema Definition (XSD) – это язык определения схем для описания и проверки структуры и содержания XML-документа. Он используется для определения элементов, атрибутов и типов данных, которые может содержать документ. Информация в XSD помогает проверить, соответствует ли описание каждому элементу, атрибуту или типу данных в документе. – *Прим. ред.*

зации движка делегируют эти одноразовые возможности другим библиотекам.

Мы надеемся, что абстракции `engine` и `collection` позволят вам легко добавить поддержку вашего любимого движка, а также потенциально внести его в кодовую базу книги, чтобы принести пользу более широкому сообществу читателей и практиков поиска на основе ИИ. Удачного поиска!

Предметный указатель

А

агрегированные сигналы 49
аддитивный бустинг 111
активное обучение 21, 47, 392, 417
алгоритм RRF 557, 560
алгоритм RSF 564
алгоритмические предубеждения 367
альтернативные метки 77, 78, 150, 205, 214
априорное распределение 383

Б

базовые модели 16, 19, 38, 52, 54, 421, 519, 520, 522, 526, 534, 545, 569
бета-распределение 380, 382, 383, 384, 388, 389
бинарная квантизация 462
бинарная классификация 335, 339
большие языковые
 модели 14, 19, 32, 66, 534
бустинг 46, 81, 107, 108, 111, 114, 121, 132, 136, 139, 140, 142, 249, 251, 252, 269, 270, 273, 276, 277, 299, 303, 546
бустинг сигналов 257

В

векторное пространство 72, 90, 304, 335, 424, 446, 452, 472, 546, 547
векторные базы данных 20, 22, 30, 55, 276, 530, 566, 578
внешний ключ 62, 64

Г

галлюцинации 33
генеративные модели 421, 526, 528, 539
генеративные системы 35

генеративный поиск 21, 43, 52, 521, 527, 529
генерация, дополненная
 результатами поиска 521
генерация, дополненная
 результатами поиска 33
генерация нового контента 53
генерация ответов на вопросы 53
генерация с дополненной выборкой 529
генерация синтетических обучающих
 данных 54, 535
гибридный поиск 19, 422, 557, 561, 562
гипотеза распределения 72, 164
глубокие нейронные сети 51, 85, 150
глубокое обучение 35, 54, 66, 419
граф знаний 43, 45, 67, 77, 79, 85, 150, 151, 152, 155, 159, 160, 170, 178, 179, 180, 181, 205, 214, 217, 444, 546
граф отношений 66, 67, 68
графы знаний 16, 19, 34, 37, 45, 46, 57, 77, 79, 149, 150, 152, 169, 179, 180, 182, 237, 535, 546, 554

Д

датафрейм 210, 294, 295, 296, 298, 314, 394, 450, 498, 504, 579
динамическая байесовская сеть 373
дисконтированный кумулятивный
 прирост 396

Е

естественный язык 37, 545

И

идеальные результаты поиска 365
извлечение ответа на вопрос 53

измерения 57, 72, 73, 74, 76, 88, 90, 99,
102, 120, 128, 279, 280, 297, 304, 305,
309, 311, 313, 335, 407, 427, 456, 462,
476, 488, 505, 523, 531, 539, 540, 543
ИИ «черного ящика» 54
инвертированный индекс 40, 67, 93, 161,
164, 165, 426, 439, 485, 557
инженерия релевантности 47

К

квантизация 19, 452, 462, 463, 464, 468,
472, 479, 485
классификаторы ранжирования 37, 50, 142
классификация намерений запросов 37
кластеризация документов 37
контролируемый ИИ 54
корпус 69, 70
косинусное сходство 75, 90, 94, 95, 102,
218, 312, 313, 426, 428, 429, 430, 431, 449

Л

латентные признаки 279, 292, 293, 304,
305, 548
лексический поиск 73, 218, 221, 223, 240,
439, 461, 557, 559, 560, 561, 564, 565,
566, 578
лемматизация 70

М

маскированное моделирование языка 433
матрёшка 463
матричная факторизация 50, 139, 287
машина опорных векторов 335
машинное обучение 35, 36, 71, 84, 119,
131, 322, 323, 326, 327, 334, 335, 345
метод грубой силы 449
метод наименьших квадратов 292
метод скользящего окна 508
модели бустинга сигналов 21, 37, 56, 120,
132, 134, 135, 137, 140, 217, 218, 252,
253, 254, 256, 258, 259, 260, 261, 262,
264, 268, 269, 277, 279
модели кликов 50, 361, 362, 363, 364, 365,
367, 368, 371, 373, 379, 380, 383, 387,
388, 389, 392, 395, 398, 402, 403, 404, 417
модели обучения
ранжированию 56, 357, 483, 484
модели персонализации 42, 56, 555, 556
модели совместной фильтрации 37, 568
мультимодальные рекомендательные
системы 41, 286
мультимодальный поиск 21, 34, 38, 76,
422, 521, 545, 546, 552
мультипликативный бустинг 111

Н

набор документов 70, 76, 131, 164, 166, 516
намерение запроса 77, 82, 120, 182, 214, 222
намерение пользователя 38, 44, 45, 147, 182
неалгоритмические
предубеждения 367, 368
неструктурированные
данные 59, 60, 61, 63, 66, 68, 79
нечеткие внешние ключи 159

О

обобщаемая функция ранжирования 56
обобщение результатов
с использованием LLM 54
обобщенная
релевантность 50, 132, 249, 251
обучение ранжированию 16, 19, 21, 34,
37, 48, 50, 110, 121, 132, 140, 142, 249,
251, 264, 265, 322, 323, 358, 387, 389,
546, 554
онтология 77, 79
отраженный интеллект 21, 26, 47, 130,
131, 355, 419

П

персонализированная релевантность 50,
132, 138, 140, 249, 251
персонализированный поиск 16, 21, 42,
45, 82, 279, 280, 282, 321, 554
плотное векторное представление 90,
91, 93, 452
плотный векторный поиск 16, 21, 43, 54,
225, 422, 423, 424, 432, 456, 459, 461,
529, 557
поведенческие
сигналы 37, 283, 285, 360, 402
поисковая система 32, 35, 39, 40, 47, 50,
66, 73, 74, 78, 81, 108, 119, 124, 126,
129, 130, 136, 151, 164, 168, 182, 186,
205, 206, 222, 229, 273, 279, 281, 282,
304, 319, 330, 350, 364, 365, 369, 371,
391, 428, 460, 464, 494, 526, 527, 532
поисковый движок 35, 39
поиск по ключевым словам 15, 42, 44, 45,
46, 57, 136, 222, 240, 280, 282, 302, 311,
313, 438
поиск по плотным векторам 421, 427
поиск по разреженным векторам 426
поиск по эмбедингам 43
показатель кликабельности 364
поле 69
полисемия 65, 66, 80
понимание домена 44, 45, 81, 278

понимание контента 44, 47, 120, 278, 280
понимание пользователя 44, 45, 120, 278, 280, 282
популяризированная релевантность 49, 138, 140, 249
последовательности символов 68, 70, 159
последовательность терминов 68, 75, 166
правила ETL 55
предпочтения персонализации 56
предубеждение 367, 368, 369, 370, 371, 372, 373, 377, 379, 389, 402, 403, 404
предубеждение
 представления 140, 403, 404
преобразователь конечного состояния 225
приближенный поиск ближайшего соседа 451
признаки 72, 76, 91, 93, 94, 110, 114, 287, 288, 293, 305, 326, 327, 328, 329, 330, 331, 333, 335, 337, 338, 339, 344, 347, 348, 354, 355, 362, 396, 417, 419, 430, 484
продуктовая квантизация 472
пулирование 440

Р

разреженное векторное
 представление 91, 93, 240, 426
разрешение сущности 64
ранг 119, 196, 293, 369, 370, 374, 393, 396
ранжирование TF-IDF 102
ранжирование с машинным обучением 140
распределительная семантика 70, 79
рекомендательные движки 40
рекомендательные системы 39, 40, 41, 279, 280, 282, 321
рекомендательные системы на основе контента 41
рекомендательные системы на основе поведения 41
рекомендации 16, 32, 34, 37, 40, 42, 43, 45, 56, 57, 123, 131, 138, 139, 140, 142, 143, 175, 217, 279, 280, 282, 283, 284, 286, 287, 291, 292, 297, 298, 299, 307, 320, 321, 424, 566
рекомендации, управляемые пользователем 42

С

сбалансированная функция
 ранжирования релевантности 100
семантические графы 16, 50, 150
семантический анализ 43, 152, 241

семантический граф знаний 150, 159
семантический поиск 16, 21, 37, 45, 50, 219, 240, 241, 306, 421, 456, 484, 485, 545, 554, 561
семантический триплет 153
сигналы 15, 56, 121, 180, 191, 215, 252, 268, 269, 271, 277, 279, 287, 288, 311, 320, 398, 403, 417
символы 68, 157, 463
синонимы 55, 59, 77, 78, 84, 132, 144, 150, 156, 158, 160, 179, 217
скалярная квантизация 462
совместная фильтрация 50, 121, 138, 139, 140, 175, 249, 251, 285, 286, 287, 301, 303
стемминг 70, 255
суждения 50, 142, 326, 333, 356, 360, 361, 362, 364, 365, 367, 370, 377, 380, 381, 387, 388, 389, 391, 394, 398, 402, 414, 539
суммаризация результатов 53
сущность 45, 62, 64, 65, 150, 204, 228, 232, 236, 284

Т

таксономия 77
термины замены 78
термины расширения 78, 172
токен 64, 81, 243, 488, 489, 491, 526, 531
трансформеры 21, 51, 423, 432, 433, 434, 439, 486, 497, 500, 510, 546
триплет RDF 153
триплеты RDF 55

Ф

фасетирование 40, 78, 163
фасеты 50, 162
фразы 44, 45, 55, 60, 71, 78, 80, 81, 82, 89, 90, 109, 132, 153, 157, 158, 159, 171, 179, 203, 225, 304, 472, 526, 558
функция косинусного сходства 74
функция полураспада 266
функция потерь 512
функция ранжирования 313, 323, 324, 347

Ц

центроид кластера 312
циклы обратной связи 47, 48, 49, 50

Ч

частота термина 94

Ш

шаблоны Херста 156

Э

эмбеddинг 38, 51, 72, 102, 216, 279

эмбеddинг векторов плотности 74

А

A/B-тестирование 21, 46, 390, 391, 392

ALS 292, 293, 294, 295, 296, 297, 303

alt. labels 77

Apache Jena 152

Apache Lucene 98, 102, 453

Apache Solr 14, 20, 22, 24, 26, 55, 102, 108,
112, 152, 163, 167, 206, 226, 229, 570,
572, 576

Apache Spark 56, 58, 570

В

backend 56

BM25 102, 103, 104, 106, 107, 110, 112,
114, 118, 119, 142, 147, 164, 218, 225,
325, 328, 330, 338, 352, 353, 356, 426,
437, 461, 494, 516, 557

С

characters 68

character sequences 68

ChatGPT 32, 34, 423, 556

Click-Through Rate 364

CLIP 548, 549

corpus 208

CTR 364, 365, 368, 369, 370, 371, 372,
373, 377

Д

DBN 373

DBpedia 151, 152

dimensions 72

Docker 22, 23, 24, 88, 89, 294, 500, 506,
570, 571, 572, 573, 575, 577

Е

edismax Solr 109

Elasticsearch 14, 20, 98, 102, 115, 162, 206,
275, 276, 325, 330, 453, 493, 578

entity resilution 64

ETL 55

Р

FAISS 461, 465, 466, 470, 474, 477, 480

feature 72, 93, 340, 344, 345, 347, 395, 396,
397, 405, 409, 441

foreign key 62

frontend 56

FST 225, 226, 229

Р

graph of relationships 67

Н

HNSW 452, 453, 459

Р

judgments 328, 338, 538

Jupyter 22, 23, 58, 88, 89, 106, 220, 325,
499, 506, 570, 571, 572, 573, 574, 577

Л

lemmatization 70

Llama 32

LLM 14, 16, 32, 51, 66, 304, 421, 486, 520

М

Mixtral 32

Н

NMSLIB 453, 457, 459, 460, 465

NumPy 461

О

OpenAI 32, 34, 423, 524, 532, 541, 548, 556

OpenSearch 20, 55, 98, 102, 115, 162, 206,
244, 275, 276, 325, 330, 453, 493, 572,
576, 577, 578

Р

Page Rank 144

phrases 68, 458

polysemy 65

POS 55

Р

RAG 19, 21, 29, 33, 38, 52, 54, 57, 422, 485,
520, 521, 529, 530, 531, 532, 543, 544,
545, 567

RDF 55, 152, 155

REBEL 153

RetroTech 123, 124, 125, 126, 127, 133,
204, 252, 259, 261, 264, 265, 288, 293,
305, 363, 367, 369, 387, 405

С

SDBN 373, 375, 376, 377, 378, 379, 380,
383, 384, 385, 386, 387, 388, 392, 393,
394, 395, 414

SKG 50, 159, 160, 161, 163, 164, 167, 168,
169, 170, 173, 175, 176, 178, 179, 180,
181, 182, 183, 185, 186, 187, 190, 191,
203, 214, 218, 221, 223, 232, 235, 238,
239, 240, 241, 242, 243

SPARQL 152

SPLADE 169, 241, 242, 243

SQL 20, 61, 62, 63, 135, 263, 579

stemming 70

SVMrank 335, 337, 339, 340, 346, 356

T

TF-IDF 101, 102, 103, 106, 118, 119, 147,
164

transformer 66

Triples 152

U

UniRel 153

V

Vespa 325, 453, 530, 578

W

Weaviate 325, 453, 576, 578

Y

Yago 152

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:
Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. **+7 (499) 322-19-38**,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ.

Трей Грейнджер, Дуг Тернбулл, Макс Ирвин

Поиск на основе искусственного интеллекта

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Люско И. Л.*
Корректор *Абросимова Л. А.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 47,61. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Современный поиск — это намного больше, чем простое сопоставление ключевых слов. Интеллектуальный поиск, который учится на взаимодействиях с пользователем, интерпретирует его намерения и использует инструменты ИИ, такие как большие языковые модели (LLM), предоставляет более точные и релевантные результаты. Эта книга научит вас создавать поисковые системы, которые понимают естественный язык и автоматически улучшаются по мере использования. Ознакомившись с десятками интересных практических примеров, вы освоите такие мощные методы на основе ИИ, как семантический поиск по эмбедам, системы вопрос-ответ на основе LLM, персонализация в реальном времени и генерация, дополненная поиском (RAG).

Издание адресовано широкому кругу читателей, знакомых с основными принципами работы поисковых систем и машинного обучения.

Среди рассматриваемых тем:

- разреженный лексический и семантический поиск на основе эмбедингов;
- системы вопрос-ответ, RAG и суммаризация с использованием LLM;
- персонализированный поиск и модели бустинга сигналов;
- обучение ранжированию, мультимодальный и гибридный поиск.

Трей Грейнджер — основатель Searchkernel, бывший главный специалист по алгоритмам и старший вице-президент по инжинирингу в Lucidworks. **Дуг Тернбулл** — главный инженер в Reddit и бывший ведущий инженер по релевантности в Spotify. **Макс Ирвин** — основатель Max.io и бывший управляющий консультант в OpenSource Connections.

«Эта книга должна быть на полке у каждого специалиста по поиску!»

Халиф Аль-Джадда, Google

«Воистину, это карта Острова Сокровищ! Вы держите в руках бесценный клад знаний о семантическом поиске».

Марк Мойу, NVIDIA

«Современные и всеобъемлющие знания, которых достаточно для создания поисковых систем мирового класса».

Кельвин Тан, SearchStax

«Начните строить собственную поисковую систему на основе ИИ с помощью простых для понимания примеров. Очень своевременное издание».

Дэвид Меза, NASA

